GUIDE TO CL-PROTOS:
AN EXEMPLAR-BASED
LEARNING APPRENTICE

DANIEL L. DVORAK*

SEPTEMBER 1988     AI88-87

* This work was supported through the AT&T Doctoral Support Program.
Abstract

CL-Protos is a machine learning program which acquires knowledge for performing expert heuristic classification. The program learns as a byproduct of performing classification under the guidance of a human teacher. When presented with a case to be classified, CL-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 of the correct classification. The program learns by selectively retaining cases and their explanations. By learning from explanations in addition to examples, CL-Protos is able to learn to classify from little training and to justify its conclusions. CL-Protos is a COMMON LISP reconstruction of the Protos “exemplar-based learning apprentice”. Protos was originally conceived by E. Ray Bareiss and Bruce W. Porter, implemented in Prolog, and applied to the domain of clinical audiology.

This guide is in two parts. The first part overviews the Protos paradigm and explains how to use the CL-Protos program. The second part is a guide to the software; it describes the high-level design of CL-Protos in terms of the major structures and functions.
Contents

I CL-Protos User's Guide 1

1 Introduction 3
  1.1 History 3
  1.2 Conditions 4
  1.3 Contacts 4
  1.4 Extensions 4
  1.5 Credits 6

2 Quick Overview of Protos 7
  2.1 Level 1 — Overview 7
  2.2 Level 2 — Classification 8
  2.3 Level 3 — Learning 9
  2.4 Glossary 10

3 The Explanation Language 12
  3.1 Terms 12
  3.2 Explanations 13
  3.3 Relations and Quantifiers 14
  3.4 Extending the Language 14
  3.5 Special Predicates 15
  3.6 Qualitative Math Relations 16

4 Tips for a Teacher of Protos 18
  4.1 Background 18
  4.2 Tips 19

5 Installing & Running CL-Protos 22
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Installation</td>
<td>22</td>
</tr>
<tr>
<td>5.2</td>
<td>Loading</td>
<td>23</td>
</tr>
<tr>
<td>5.3</td>
<td>Running CL-Protos</td>
<td>24</td>
</tr>
<tr>
<td>5.4</td>
<td>Sample Dialogue — Entering a new case</td>
<td>25</td>
</tr>
<tr>
<td>5.5</td>
<td>Escapes</td>
<td>31</td>
</tr>
<tr>
<td>5.6</td>
<td>Control Menu</td>
<td>31</td>
</tr>
<tr>
<td>5.7</td>
<td>Transformations</td>
<td>33</td>
</tr>
<tr>
<td>5.8</td>
<td>KB Files</td>
<td>33</td>
</tr>
<tr>
<td>5.9</td>
<td>Terminal Input Idiosyncrasies</td>
<td>34</td>
</tr>
<tr>
<td>5.10</td>
<td>Miscellaneous</td>
<td>35</td>
</tr>
</tbody>
</table>

**II** CL-Protos Software Guide

6 Introduction

6.1 Related Documents

6.2 Design

6.3 CL-Protos Source Code

6.4 Source Code Conventions

7 Global Objects

7.1 Numeric Constants

7.2 Switches

7.3 Anchors

7.4 Menus

7.5 Verbs & Quantifiers

7.6 Category Network

8 Structures

8.1 Terms

8.2 Predicates

8.3 Relational Links

8.4 Difference Links

8.5 Explanations & Matches

8.6 KBPM Search Graph

8.7 Other
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.8</td>
<td>Non-Structures</td>
<td>48</td>
</tr>
<tr>
<td>9</td>
<td>Functions</td>
<td>49</td>
</tr>
<tr>
<td>9.1</td>
<td>Protos Classification Algorithm</td>
<td>49</td>
</tr>
<tr>
<td>9.2</td>
<td>Who Calls Who</td>
<td>49</td>
</tr>
<tr>
<td>9.2.1</td>
<td>protos</td>
<td>50</td>
</tr>
<tr>
<td>9.2.2</td>
<td>classify</td>
<td>50</td>
</tr>
<tr>
<td>9.2.3</td>
<td>get-raw-reminding</td>
<td>51</td>
</tr>
<tr>
<td>9.2.4</td>
<td>combine-reminding</td>
<td>51</td>
</tr>
<tr>
<td>9.2.5</td>
<td>build-hypotheses</td>
<td>51</td>
</tr>
<tr>
<td>9.2.6</td>
<td>test-hypotheses</td>
<td>52</td>
</tr>
<tr>
<td>9.2.7</td>
<td>discuss-match</td>
<td>52</td>
</tr>
<tr>
<td>9.2.8</td>
<td>discuss-success</td>
<td>53</td>
</tr>
<tr>
<td>9.2.9</td>
<td>discuss-failure</td>
<td>54</td>
</tr>
<tr>
<td>9.2.10</td>
<td>add-new-exemplar</td>
<td>54</td>
</tr>
<tr>
<td>9.2.11</td>
<td>discuss-explanation</td>
<td>55</td>
</tr>
<tr>
<td>9.2.12</td>
<td>compare-new-case</td>
<td>56</td>
</tr>
<tr>
<td>9.2.13</td>
<td>explore-differences</td>
<td>56</td>
</tr>
<tr>
<td>9.2.14</td>
<td>get-explanation</td>
<td>56</td>
</tr>
<tr>
<td>9.2.15</td>
<td>install-relation</td>
<td>57</td>
</tr>
<tr>
<td>9.2.16</td>
<td>kbpms</td>
<td>57</td>
</tr>
<tr>
<td>9.2.17</td>
<td>reassess-reminding</td>
<td>60</td>
</tr>
<tr>
<td>9.2.18</td>
<td>check-term-name</td>
<td>60</td>
</tr>
<tr>
<td>9.2.19</td>
<td>getname</td>
<td>61</td>
</tr>
<tr>
<td>10</td>
<td>Source Files</td>
<td>62</td>
</tr>
<tr>
<td>10.1</td>
<td>CL-Protos Source Files</td>
<td>62</td>
</tr>
<tr>
<td>10.2</td>
<td>&quot;System&quot; Files</td>
<td>63</td>
</tr>
<tr>
<td>11</td>
<td>Miscellaneous</td>
<td>65</td>
</tr>
<tr>
<td>11.1</td>
<td>Extending the Explanation Language</td>
<td>65</td>
</tr>
<tr>
<td>11.2</td>
<td>Dark Corners</td>
<td>66</td>
</tr>
</tbody>
</table>
Part I

CL-Protos User’s Guide
Chapter 1

Introduction

This is a user's guide to CL-Protos, a computer program that learns as a byproduct of performing classification under the guidance of a human teacher. CL-Protos is based on the Protos "exemplar-based learning apprentice" conceived by E. Ray Bareiss and Professor Bruce W. Porter at the Artificial Intelligence Laboratory of The University of Texas at Austin. Protos\(^1\) is of potential interest to researchers interested in automated knowledge acquisition, machine learning, and case-based reasoning. CL-Protos is an implementation of the Protos paradigm in COMMON LISP.

This guide is intended for users of the CL-Protos program. In addition to explaining how to install and run CL-Protos, it also includes a simple overview of the Protos paradigm and offers tips for the user/teacher of Protos.

1.1 History

The published research results on Protos are based on experiments performed with the original research implementation of Protos, developed by Ray Bareiss and Bruce Porter in 1986–1987. This program was implemented in Prolog and C, with a user interface tied to Sun workstations. CL-Protos is a reconstruction of the Protos paradigm based primarily on the design description in Ray Bareiss's doctoral dissertation [Bar88]. CL-Protos is implemented entirely in COMMON LISP and is quite portable, having run on Symbolics, TI Explorer, IIP 9000, Sun 3, and Macintosh machines. As a reconstruction, CL-Protos benefited from lessons learned during the Protos research — the resulting implementation incorporates some extensions suggested by Ray Bareiss (see section 1.4). Since CL-Protos is expected to serve as a base for further research, we have attempted to make the implementation well-organized and well-commented.

\(^1\)In this guide the term Protos refers to the paradigm of the exemplar-based learning apprentice and, in a few instances, the original Prolog/C implementation. The term CL-Protos refers specifically to the COMMON LISP implementation of Protos.
1.2 Conditions

The CL-Protos program is freely distributed to researchers interested in automated knowledge acquisition, machine learning, and case-based reasoning. The program is a research tool, not a product, so we cannot give any warranty that the program is suitable for a given application nor that it is free of bugs. CL-Protos is distributed as a courtesy among researchers. Copyright and all commercial rights are reserved.

We have attempted to make this implementation readable and well-commented. You are welcome to build upon and modify CL-Protos for your own research. If you find and/or fix any bugs, please let us know. Also, if you enhance CL-Protos in a way that might be of general interest, please let us know — we may want to put those enhancements into the “official” version. We’d also appreciate hearing about any noteworthy uses of the software.

1.3 Contacts

Questions and comments about CL-Protos may be directed to Dan Dvorak, Ray Bareiss, or Bruce Porter. At present, Dan Dvorak will be best able to answer questions about implementation details (and bugs) of CL-Protos. Addresses are shown below.

Bruce W. Porter
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
porter@cs.utexas.edu
(512) 471-9563

Daniel L. Dvorak
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
dvorak@cs.utexas.edu
(512) 471-9576

E. Ray Bareiss
Department of Computer Science
Box 115, Station B
Vanderbilt University
Nashville, TN 37235
bareiss@vuse.vanderbilt.edu
(615) 322-3233

Although there will be some further development of CL-Protos at UT-Austin, Ray Bareiss will maintain the “official” version of CL-Protos at Vanderbilt.

1.4 Extensions

CL-Protos has some extensions beyond the Protos paradigm described in [Bar88]. The main extensions are:
1. The procedure for combining remindings checks to see if any pair of combined remindings share a common specialization. If so, the common specialization inherits the strength of the remindings to the two categories. Thus, the common-specialization category will be tried first, whereas, without this extension, it might not be tried at all.

2. A new function “trim-remindings” guards against the possibility of diffuse remindings, i.e., a feature which has too many remindings associated with it. Such a feature can cause too many different hypotheses to be tried. This function removes the weakest remindings associated with the feature.

3. The number of exemplars selected from a hypothesized category is no longer a constant number; the number is now controlled by the relative strength of the combined reminding. Thus, several exemplars will be considered from a strongly reminded category, whereas only the prototype will be considered from a weakly reminded category.

4. In addition to the normal learning mode wherein Protos asks the teacher about every match it performs, CL-Protos provides a “performance mode” wherein it does all the matching first and then presents the matches in decreasing order of similarity, without teacher involvement. This is useful after Protos has been trained in a given subject and is to be tested and/or used as a heuristic classifier.

5. An option in the control menu alters the similarity metric so that CL-Protos computes the N-th root of the overall similarity of a match (where N is the number of exemplar features). Thus, overall similarity equals the “average” strength of the feature-to-feature matches, and therefore an exemplar with many features will not be penalized relative to an exemplar with fewer features.

6. An experimental extension to the Protos explanation language provides some new relations that support simple qualitative mathematical reasoning, in the style of Kuipers. See section 3.6.

7. Quantitative-to-qualitative transformations can be defined on selected features so that they will be transformed when they appear in a new case. For example, (temperature 100) could be transformed into (fever mild). See section 5.7.

8. An extension that was begun but never completed is a “model of medical diagnosis”. The intent was that terms should be identified according to type (patient history, lab data, diagnostic category, etc.) and that information would be used in seeking and/or filtering explanations. It’s still a good idea, but the student who started it eventually chose a different project. The existing code can be switched on or off from the control menu, but all it currently does is prompt for the type of each new term.
1.5 Credits

CL-Protos (Release 1.0) was developed at the Artificial Intelligence Lab of The University of Texas at Austin. The development was a group effort involving several individuals:

- Rita Duran developed and tested the knowledge-based pattern matcher (kbpm.lisp) and associated heuristics (heuristic.lisp).
- Ben Tso developed and documented the parser (parse.lisp).
- Jim Kroger developed an early version of functions for saving and loading knowledge bases.
- Hilel Swerdlin performed numerous tests on early versions of CL-Protos, developed the code for merging a case with an exemplar and the code to control the number of exemplars chosen based on the strength of the combined reminding. Hilel also wrote portions of an early draft of the software guide.
- Agnar Aamodt developed the code for transformation of parameter values, such as for quantitative-to-qualitative transformation (transform.lisp).
- Dan Dvorak organized the reconstruction effort, architected the overall design in COMMON LISP, and programmed everything else not specifically credited above.
- Ray Bareiss answered numerous questions about Protos, suggested several improvements over the original program, and performed some testing on CL-Protos.
Chapter 2

Quick Overview of Protos

This chapter provides a simple overview of the Protos paradigm, described in three successively deeper levels of detail. The last section is a glossary of some terms commonly used in describing the Protos paradigm. This is intended as a minimal introduction to the Protos paradigm for those who wish to use CL-Protos.

This overview omits many details of the paradigm that allow it to be an efficient classification and learning system. For a more thorough account, two other documents are highly recommended — [BPW88] provides a good general description of the design concepts, and [Bar88] provides much more detail.

2.1 Level 1 — Overview

1. CL-Protos is a machine learning program which acquires knowledge for performing expert heuristic classification. The program learns as a byproduct of performing classification under the guidance of a human teacher. When presented with a case to be classified, CL-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 of the correct classification. The program learns by selectively retaining cases and their explanations. By learning from explanations in addition to examples, CL-Protos is able to learn to classify from little training and to justify its conclusions.

2. As a paradigm, Protos combines into one unified architecture the tasks of knowledge acquisition from a domain expert, heuristic classification, and learning from problem-solving failures. It may help to think of Protos as an expert system that improves with use and learns through direct interaction with the domain expert (without requiring a "knowledge engineer" in the middle).

3. Protos is a case-based or exemplar-based classification system — it takes input in the form of a new case (i.e., a set of features) and then tries to find a similar previous 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 features of the exemplar.

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

- **Features** are terms used by the teacher to describe a case. For example, I could describe the case “1978 Honda Accord” with the features piston-engine, windshield, radio, (doors 2), hatchback, and (wheels 4). A feature can be in the form of a proposition or a predicate with arguments.

- **Exemplar** is a case which Protos has processed and retained. An exemplar is an instance of a concept. For example, the exemplar “1978 Honda Accord” is an instance of the concept “car”.

- **Category** is a collection of instances of a concept. The category “car” might consist of the exemplars “1978 Honda Accord”, “1986 Chrysler Reliant”, and “1985 Lincoln Continental”.

- **Relations** are learned from teacher-provided explanations. For example, Protos will ask what the relevance of “piston-engine” is to “car”. The teacher might respond with the explanation “piston-engine has typical generalization engine is part of car”. From this, Protos installs two relations in its knowledge base: “piston-engine has typical generalization engine” and “engine is part of car”.

5. Protos is a learning apprentice — it learns as a natural by-product of attempting to solve problems. When Protos mis-classifies a new case, it enters into a dialogue with the teacher to learn from the mistake. All of the discussion is focused within the context of the particular case. It is easier for an expert to respond to this type of focused discussion than to try to tell someone all that he or she knows about a subject.

### 2.2 Level 2 — Classification

This section describes the algorithm Protos uses to perform classification. Points about where and how Protos learns are mentioned, but details are left to Level 3.

1. The user of Protos is called a **teacher** (given that Protos itself is the learner). The teacher enters a new case described as a set of **features**. For example, the case “my-car” might be described with the features piston-engine, (wheels 4), seats, (body steel), steering-wheel, radio.

2. Each feature can evoke one or more **reminders** to possible categories. For example, (wheels 4) may evoke reminders to “car” and “wagon”; carpet may evoke reminders to “house” and “car”. Each reminding has an associated strength. For example,
piston-engine evokes a strong reminding to "car" whereas carpet evokes a weak reminding to "car".

3. The evoked remindings are combined to produce a set of hypotheses, i.e., a set of categories that are described through exemplars. From each such category Protos selects the most prototypical exemplars as candidates to match against the new case.

4. The similarity comparison is done on a feature-by-feature basis starting with the most important feature of the exemplar, trying to find how it is related to any feature of the new case. Each attempt at matching a feature results in a strength value in the range 0 - 1. If an exemplar feature is identical to a new case feature, then the strength is 1.0; if there is an explanation linking the two features, such as "piston-engine has typical generalization internal-combustion-engine has typical specialization Wankel-engine", then the strength will usually be less than 1.0. If the exemplar feature cannot be matched with any feature of the new case, then the strength value is (1.0 - importance). Thus, failure to match a very important feature (one whose importance is close to 1.0) results in a low strength value. The strength values from each match are multiplied together to form an overall similarity value. Thus, each feature-to-feature match that is less than 1.0 weakens the overall similarity.

5. If Protos is running in its "maximum learning" mode, it will present each exemplar-to-new case match as soon as it is computed. Otherwise, Protos will compute matches for all of the candidate exemplars and then present them to the teacher, ordered by decreasing overall similarity. In either case, when Protos presents a match, it asks the teacher if the matched exemplar is of the correct category (i.e., the correct classification for the new case). If the teacher says "yes", Protos then asks if the chosen exemplar is a good one for matching with the new case. The teacher can agree, or can select a different exemplar in the same category, or can decide that the new case should become a new exemplar of this category.

If the teacher says "no" (the exemplar is not of the correct category for the new case), then Protos tries to learn from the mistake. It reassesses the remindings that led it to try this incorrect category, it asks the teacher for censors (features of the case which tend to rule out the incorrect category), and it asks the teacher if any of the exemplar's unmatched features should really be considered more important than they currently are (and thus weaken future matches to this incorrect category). After responding to these questions, the teacher may then tell Protos to either reconsider the new case from the very beginning or just try the next candidate.

2.3 Level 3 — Learning

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. This section describes how Protos learns four types of indices — remindings, censors, prototypicality, and difference links.
CHAPTER 2. QUICK OVERVIEW OF PROTOS

1. Reminding 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 engine is part of car, Protos learns a reminding from engine to car. The strength of the reminding comes from the strength of the explanation. Several more details about the learning of reminding are explained in [Bar88, pages 67–71].

2. Censors are negative reminding associated with categories, and are learned from discussion with the teacher. When a classification is rejected, the teacher is asked if any of the features of the new case are mutually exclusive with the classification. For each such feature, the teacher specifies the strength of the negative evidence as weak, moderate, strong, or absolute.

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 “near miss” matches during classification. When the teacher rejects a strong match, the program remembers this (temporarily). When the case is finally classified, Protos recalls the near misses for discussion with the teacher. Of the features which did not match between the near-miss exemplar and the new case, the teacher specifies which features represent important distinctions. These are installed in a difference link between the near-miss exemplar and the matched exemplar.

2.4 Glossary

case — a featural description presented to Protos for classification.

category — a collection of instances of a concept; the extension of the concept. (The “instances” are called exemplars).

category network — Protos’ network of domain knowledge consisting of terms (i.e., features, categories, and exemplars), reminding, censors, importances, prototypicalities, relational links, and difference links.

censor — a negative reminding, i.e., an index into the category network saying that the presence of a certain feature tends to rule out a given classification.

difference link — connects two exemplars (in the same or different categories) and indicates important features which discriminate between the two. The presence of a difference link between an exemplar and a near-miss exemplar allows Protos to avoid making a plausible, but incorrect (or non-optimal), classification.

exemplar — a case which Protos has processed and retained. It is embedded in the category network as a member of a category, with explanations of the relevance of its case features to the category.
2.4. GLOSSARY

explanation — a chain of plausible inferences (relational links) which allows Protos to conclude the equivalence of two features (a feature-to-feature explanation) or that a feature is relevant to a case’s category membership (a feature-to-category explanation).

featural exemplar — a feature of a case may itself be an exemplar-containing category when viewed at a more detailed level (e.g., “engine” may be a feature of “my-car”, and “engine” may also be a category having exemplars of various specific engines.

features — the labels for the attributes of a case that comprise the teacher’s description.

importance — a measure of the necessity of a feature being present for a given classification to be made. Importance is an estimate of \( p(f|C) \), the probability of the feature's presence given the classification.

knowledge-based pattern matching — a procedure which uses domain knowledge within the category network to construct an explanation of the equivalence of a new case to an exemplar. This procedure currently performs a uniform-cost search with heuristics to promote or inhibit certain explanation steps.

learning apprentice — a system which provides interactive aid in problem-solving and learns as a natural consequence of its use (paraphrased from Mitchell).

near miss — an exemplar which was strongly, but incorrectly, matched by a new case.

prototype — the most representative exemplar of a category.

prototypicality — a measure of the centrality of exemplars within a category. An exemplar which participates often in strong, correct matches is inferred to be prototypical.

relational links — single-step inferences which relate two terms in the category network.

reminding — an index in the category network relating a feature to a category which is possibly relevant to the classification of the case when the feature is present. Reminders are compiled from feature-to-category explanations. The strength of a reminding estimates \( p(C|f) \), the probability of the classification given the feature.

term — an object, named by the teacher, which may be used as a feature, category, or exemplar, as needed. A term may be expressed as a proposition, as in headache, or as a predicate with arguments, as in (fever mild).
Chapter 3

The Explanation Language

CL-Protos learns about new terms and relations among terms from explanations entered by the teacher. These terms and relations must be expressed in the simple language defined in this chapter.

3.1 Terms

Features, categories, and exemplars are collectively known as “terms”. Terms may be expressed as a single symbol (e.g., backrest, pedestal, etc.) or as a predicate with arguments (e.g., (fever mild), (weather sunny cool), etc.). In both cases, terms must be valid Lisp expressions, either a simple symbol or a list. In general, a simple symbol in Common Lisp is anything that can’t be interpreted as a number or as a symbol in the keyword package or as a package name followed by a symbol (see [Ste84, pages 339–346] for details). The following rules will ensure that the name is a simple symbol:

- Begin the name with an alphabetic character.
- Continue the name with any alphabetic characters, digits, and any of the following: * - + = _ % $ @ ! . < > [ ] { } /.
- Do not use a colon (:), semicolon (;), comma (,), sharp-sign (#), vertical-bar (|), or any quote or accent mark anywhere in the name.
- Don’t use any of the reserved symbols of Common Lisp, such as T and NIL. (The first rule will eliminate problems with other reserved symbols such as *standard-output* which always begin with *).

Examples of simple symbols are wheels, fm-radio, and on/off_switch.

Parentheses must be used in specifying a predicate with arguments, and they must be balanced as with any Lisp list. Examples include:

(legs 4)
3.2. EXPLANATIONS

(fever mild)
(weather sunny (humidity high))

The syntax of a term is:

\[
\text{term} ::= \text{proposition} \mid \text{predicate} \\
\text{proposition} ::= \text{simple-symbol} \\
\text{predicate} ::= (\text{simple-symbol} \text{ argument*}) \\
\text{argument} ::= \text{term} \mid \text{number}
\]

3.2 Explanations

Explanations are statements that describe how one set of terms is related to another set of terms. An explanation must be in the form of a Lisp list, so it is always enclosed in parentheses. In its simplest form an explanation connects two terms, as in:

(fire causes heat)

The syntax of a 1-step explanation is:

\[
\text{explanation} ::= (\text{condition} \text{ terms } \text{ quantifiers } \text{ relation } \text{ terms}) \\
\text{condition} ::= \text{if-when category is term } \text{then} \mid \text{if-when case has-features termlist } \text{then} \mid \text{if-when exemplar has-features termlist } \text{then} \mid \\
\text{if-when} ::= \text{if} \mid \text{when} \\
\text{has-features} ::= \text{has features} \mid \text{hf} \\
\text{terms} ::= \text{term} \mid \text{termlist} \\
\text{termlist} ::= (\text{term} \text{and} \text{term}* ) \\
\text{quantifiers} ::= \text{quantifier} \mid \text{quantifier quantifiers} \\
\text{relation} ::= \text{See table 3.1} \\
\text{quantifier} ::= \text{See table 3.2}
\]

Some examples of syntactically valid explanations are:

(fire causes heat)
(drought sometimes causes famine)
((fever mild) is caused by infection)
((heat and humidity) causes muggy-weather)
(if category is cars then rubber is part of tires)
(when exemplar hf (engine and wheels) then chrome us co body)

Note the use of the word "and" to denote a conjunction in (heat and humidity). Without the word "and" the expression would be interpreted as a predicate heat with argument humidity.

Incidentally, the explanation parser in CL-Protos allows multiple 1-step explanations to be "chained" together. For example, instead of typing
Table 3.1: Relations of the Protos explanation language.

\[(\text{fire causes heat})\]
\[(\text{heat causes thermal-expansion})\]

the user could instead type

\[(\text{fire causes heat causes thermal-expansion})\]

3.3 Relations and Quantifiers

The relations and quantifiers understood by CL-Protos are listed in tables 3.1 and 3.2, respectively. Either the full name or the abbreviation of each relation and quantifier may be used during input. The abbreviations mode in the control menu determines whether the full name or abbreviation is displayed on output.

3.4 Extending the Language

The set of relations and quantifiers shown in tables 3.1 and 3.2 is adequate for a number of domains, but you may wish to add new relations and quantifiers to tailor the language for your domain. To do this, two files must be edited — verb.lisp and parse.lisp. The file verb.lisp has a structure for each verb and each quantifier containing its name and strength. The file parse.lisp contains three association lists near the front that, collectively, associate a given word or phrase with the appropriate verb or quantifier: this
### 3.5. SPECIAL PREDICATES

<table>
<thead>
<tr>
<th>Abbr.</th>
<th>Quantifier</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Context:</strong></td>
<td>al</td>
</tr>
<tr>
<td></td>
<td>us</td>
</tr>
<tr>
<td></td>
<td>st</td>
</tr>
<tr>
<td></td>
<td>occ</td>
</tr>
<tr>
<td><strong>Strength:</strong></td>
<td>str</td>
</tr>
<tr>
<td></td>
<td>mod</td>
</tr>
<tr>
<td></td>
<td>we</td>
</tr>
<tr>
<td><strong>Belief:</strong></td>
<td>cert</td>
</tr>
<tr>
<td></td>
<td>prob</td>
</tr>
<tr>
<td></td>
<td>poss</td>
</tr>
<tr>
<td><strong>Temporal:</strong></td>
<td>ins</td>
</tr>
<tr>
<td></td>
<td>qu</td>
</tr>
<tr>
<td></td>
<td>grad</td>
</tr>
</tbody>
</table>

Table 3.2: Quantifiers of the Protos explanation language.

includes recognizing abbreviations and synonyms. It should be fairly obvious from the structure of these lists as to what changes will need to be made; see section 11.1 for details.

#### 3.5 Special Predicates

There are two special predicates in the CL-Protos explanation language that are “built in” and handled specially. The first is a negation predicate, specified either as nota or no, as in (not feverish) and (no hypertension). This predicate automatically forms a mutual-exclusion relation with its single argument, as in:

```
((no hypertension) mutually exclusive with hypertension)
```

The second special predicate was added for medical history information, as in (history-of hypertension). This predicate establishes an automatic “sometimes exhibits” relation with its single argument. For example:

```
((history-of hypertension) sometimes exhibits hypertension)
```

The predicate “history-of” may be abbreviated “h-o” or “h/o”. As with all predicates, there can be nesting, as in:

```
(no (history-of hypertension))
```

For both of the special predicates, the associated relations are created at the first appearance of the predicate and argument.
3.6 Qualitative Math Relations

This section describes an extension to Protos for allowing a primitive type of qualitative mathematical reasoning, inspired by the qualitative mathematics of Kuipers [Kui86]. This is an experimental feature which may be further extended in future research. The essential ideas are these:

- A predicate with a single argument can be viewed as a variable with a value. For example, in the term \textit{(pressure normal)}, the predicate \textit{pressure} is the name of a variable and the argument \textit{normal} is its value.

- Variables can be related to each other through some simple qualitative mathematical relations. For example, the relation:

\[(\text{pressure } m^+ \text{ level})\]

expresses the fact that the fluid pressure at the bottom of a tank has a positive monotonic relation to the level of fluid in the tank, \textit{i.e.}, the higher the level the higher the pressure. The effect of this relation is that when Protos is searching through the category network it sees a very strong relation between the terms \textit{(pressure normal)} and \textit{(level normal)}. Likewise for \textit{(pressure high)} and \textit{(level high)}, and for \textit{(pressure low)} and \textit{(level low)}.

- Some relations between variables are symmetric while others are directed. For example, \textit{(pressure m+ level)} could just as well have been stated as \textit{(level m+ pressure)} — if one is high the other must be high too. However, consider the case of a car driving into the wind. Stepping on the gas makes the car go faster, but going faster does not necessarily imply that the driver is stepping on the gas (the wind might have decreased). We express these relations as \textit{(gas fm+ car-speed)} and \textit{(wind-speed fm- car-speed)}. The “f” in \textit{fm+} and \textit{fm-} indicates a \textit{forward} direction-of-effect. The inverse of \textit{(gas fm+ car-speed)} is \textit{(car-speed rm+ gas)}; the “r” in \textit{rm+} and \textit{rm-} indicates a \textit{reverse} direction-of-effect.

- In any explanation found by Protos, the directed relations must all be of the same direction, \textit{i.e.}, all forward or all reverse. Although \textit{gas fm+ car-speed rm- wind-speed} is indeed a path through the category network, it is not a valid explanation. In other words, two variables are not related just because they individually affect a third variable.
3.6. **QUALITATIVE MATH RELATIONS**

- When looking for a feature-to-feature or feature-to-category explanation, Protos traverses these mathematical relations much the same as other relations. The main difference is that it propagates a value for each variable (*e.g.*, high, normal, or low) as it searches for an explanation.

- One additional feature of the qualitative mathematical relations is that they will also propagate a “direction of change” for each variable if it is supplied as a second argument. For example, \((\text{pressure high inc})\) means that the pressure is high and increasing \((\text{inc} \equiv \text{increasing}, \text{dec} \equiv \text{decreasing}, \text{and std} \equiv \text{steady})\). If the direction of change is not specified (as in the earlier examples), then it will appear in explanation print-outs as \(\text{xxx}\), *e.g.*,

\[
((\text{pressure high xxx}) \text{ m}^+ (\text{level high xxx}))
\]

Table 3.3 shows the current qualitative math relations and their inverses. The monotonically decreasing relations (\(\text{m}^-, \text{fm}^-, \text{and rm}^-\)) need to know how to invert values and directions of change. They use the following tables of inverses:

<table>
<thead>
<tr>
<th>normal</th>
<th>normal</th>
<th>inc</th>
<th>dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>mildly-elevated</td>
<td>mildly-depressed</td>
<td>std</td>
<td>std</td>
</tr>
<tr>
<td>moderately-elevated</td>
<td>moderately-depressed</td>
<td>dec</td>
<td>inc</td>
</tr>
<tr>
<td>severely-elevated</td>
<td>severely-depressed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>profoundly-elevated</td>
<td>profoundly-depressed</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

There are a few things to know if you plan to use the qualitative math relations:

1. You may wish to define abbreviations and/or synonyms for the above value names. For example, if high/normal/low is sufficiently expressive for your application, then define high as an abbreviation (or synonym) for mildly-elevated and low for mildly-depressed.

2. If you want Protos to recognize that there is an approximate equivalence between “adjacent” values, then you must tell it. For example, the relation:

\[(\text{mildly-elevated sometimes equivalent moderately-elevated})\]

would allow Protos to find a match between \((\text{pressure mildly-elevated})\) and \((\text{level moderately-elevated})\).

3. In order for Protos to know that a word such as \text{pressure} is a variable (*i.e.*, a predicate, not a term), the first appearance of the word must be in the form of a predicate with argument(s), *e.g.*, \((\text{pressure normal})\), such as when entering a feature of a case. Once this is done, then Protos will accept qualitative math relations for \text{pressure}. 
Chapter 4

Tips for a Teacher of Protos

Protos is an exemplar-based learning apprentice — it learns from examples and explanations provided by a teacher. As in any student–teacher relationship, the teaching is most effective when the teacher has a good mental model of what the student knows and how the student thinks. These notes are for a teacher of Protos; they describe what Protos knows and how it reasons. It is quite important for the teacher to form a correct mental model of Protos because if it's wrong it is difficult to correct and results in less-than-ideal instruction to Protos.

4.1 Background

1. First, to gain an overall understanding of how Protos works, read Chapter 2 or one of the overview articles on Protos such as [BPW88]. Some key points to remember about Protos are:

   - Protos learns from examples and from explanations.
   - By retaining exemplars that are prototypical of a given category, Protos can recognize new examples of the category through a similarity match.
   - Through teacher-provided explanations of how features are related to each other, Protos gains the knowledge needed to recognize similarities between non-identical features.
   - Protos always focuses its discussion with the teacher in the context of trying to match a new case to an existing exemplar.

2. In some ways Protos is like a child — it begins life knowing basically nothing about the subject that you are going to teach it, but it has a good memory and learns exactly (and only) what you teach it. Be aware that Protos is very trusting of your teaching — if you tell it that snow causes sand-storms it will believe you and it will never forget it (unless told to).
3. Although Protos will appear to make intelligent decisions about classifying new cases, it can also make silly "mistakes" simply because it has no body of common sense about the subject you are instructing it in. If you tell Protos that plants require water and then later tell it that water has typical specialization ice, Protos will then be capable of finding what it believes to be a perfectly reasonable association between plants and ice.

4.2 Tips

Organize your thoughts. Think carefully about what features and categories you will use to organize and present your subject to Protos. Lay them out on paper and show how the categories are related to each other through generalization, functional, causal, and part-to-whole hierarchies. Decide which categories will be described through exemplars. Decide what features will be used to describe the various exemplars. Think about how you will explain to Protos the relation of each feature to a category. This is a fair amount of up-front work, but is very important to help keep inconsistencies from creeping into the knowledge base. For example, without prior organization you might say dog has part fur and then later say cat has typical generalization mammal, mammal has part fur.

Present the best examples first. When presenting examples of a given category to Protos, it is best to present the most prototypical case first and work your way down to the outlying examples. This ordering will ensure that the most prototypical example of a category is saved as its own exemplar rather than being merged with some less prototypical example, and it will generally result in fewer exemplars being created. For example, if you are entering examples of the category cars, it would be best to start with a "typical" car like the 1986 Chrysler Reliant (4-door, metal body, piston engine) before entering a speciality car like the 1984 Mazda RX-7 (2-seater, Wankel engine, aluminum wheel rims).

English first, then translate. When Protos asks you how X is related to Y, explain it in English first using vocabulary that is natural to you, then translate it into Protos' explanation language. Don't try to formulate explanations directly in Protos' language! Past experience has shown that the former approach yields much better explanations. This is why, when Protos asks for an explanation, it first asks you to enter your thoughts in English.

Give educational explanations. When you give an explanation relating two terms, it is tempting to just give a correlational explanation (i.e., using verbs such as co-occurs, requires, implies, suggests). These verbs merely state that one feature is statistically related to another. Before you give a correlational explanation, look through the other types of relations (definitional, generalization/specialization, causal, functional...).

---

1Nonsensical explanations such as plants require water has typical specialization ice can be avoided by attaching conditions to individual relations so that they will be used only in suitable contexts.
to-whole) to see if you can give a stronger, more educational explanation. For example, it is more educational to say "windshield is part of cars" than to say "windshield co-occurs with cars". Likewise, it is more informative to say "Wankel-engine has typical generalization internal-combustion-engine is part of cars" than to say "Wankel-engine suggests cars".

**Use appropriate detail.** The preceding tip recommends educational explanations, but don’t take this idea too far. When you are formulating an explanation of the relation between A and Z, you might feel compelled to enter a very educational explanation such as "A implies B causes C has function D suggests E has typical generalization F is consistent with Z". That’s fine if all of those intermediate terms are needed for the level of detail that you want, but be aware that the search space that Protos must explore explodes exponentially with the amount of detail in the category network, so detailed explanations can make Protos slow.

**Does this increase my belief?** When Protos asks you to explain how a feature of a case is related to its classification (i.e., its category), always ask yourself "does the presence of this feature increase my belief in the classification?" If the answer is "no" or "just a little", then there is no need to enter an explanation. Protos will treat unexplained features as spurious. For example, the feature carpet is often found in cars, but it is hardly a criterial feature, and might therefore be left unexplained. The original reason for this tip was that if Protos is given a lot of weak feature-to-category explanations, it generates too many reminders (and that slows it down). However, CL-Protos now limits the maximum number of reminders that a given feature can possess, so don’t be too concerned about this tip.

**Attach conditions.** Remember that you can attach conditions to explanations to make them valid only in a specific context. This is necessary to prevent "silly" explanations such as rubber is part of wheels, wheels is part of oxcarts (oxcarts have wooden wheels). To prevent this, you can say if category is cars then rubber is part of wheels.

**Feature-to-category explanations are the most important.** Protos gleans reminders from feature-to-category explanations and estimates importances from category-to-feature explanations. The \( F \rightarrow C \) explanations are by far the more important since it is the reminders that suggest categories for matching. The \( C \rightarrow F \) explanations are less important since importances only need to be approximate to work. So, when Protos asks you how a feature is related to a category, it is best to give an \( F \rightarrow C \) explanation. (You can give a \( C \rightarrow F \) explanation, but when Protos inverts it into an \( F \rightarrow C \) explanation, its strength may differ from what you had intended.)

**Feature-to-exemplar explanations: rare but useful.** Be aware that you can give Protos an explanation that relates a feature directly to a specific exemplar. These are somewhat rare, but when you have a feature that is idiosyncratic to a particular exemplar, such an explanation helps Protos to immediately recall that exemplar. For example, "leaky-trunk suggests my-old-car".
Multiple antecedents and consequents. CL-Protos allows multiple antecedents and multiple consequents in an explanation. For example, you can say "(heat and high-humidity) causes muggy-weather". Also, you may enter an explanation in two or more parts, as in: "sunlight causes heat, rain causes high-humidity, (heat and high-humidity) causes muggy-weather".

Something to be careful about here is that the word "and" is used as a term of logic. So, saying \((A \text{ and } B) \text{ enables } C\) means that if \(A\) and \(B\) are both true, then \(C\) is enabled. This is not the same as saying \(A\) enables \(C\) and \(B\) enables \(C\). Likewise, saying \(X\) causes \((Y \text{ and } Z)\) is not the same as saying \(X\) causes \(Y\), \(X\) causes \(Z\).

Add new verbs and quantifiers. There is nothing sacred about the verbs and quantifiers in the explanation language of CL-Protos. If you find that there are other verbs and quantifiers that you wish to use in expressing explanations, then add them to Protos. The changes will be in verb.lisp and parse.lisp.

Run a small experiment. Before you begin using Protos for your intended task, develop a small "throw-away" knowledge base of at least 10 cases spread over 2-3 categories in the domain of your intended task. Even better would be 10 categories with 3-5 cases each. This has several benefits: it familiarizes you with how to interact with Protos, it helps you decide the appropriate terms for describing cases, you may discover new quantifiers and verbs to add to the explanation language, and you can experiment without fear of messing up something important.
Chapter 5

Installing & Running CL-Protos

This chapter describes how to use CL-Protos. It explains how to invoke the program, describes the purpose of various modes and switches, and displays portions of a sample dialogue between CL-Protos and a user.

5.1 Installation

The first step in installing CL-Protos is to load the distribution tape into your system. Symbolics tapes will be in carry format and can be read by saying:

\[
\text{(tape:carry-load)}
\]

UNIX\(^1\) tapes will be in tar format and can be read with the command

\[
\text{tar -xvf /dev/rst0} \^{2}
\]

The tape contains the source code in files named *lisp, i.e., global.lisp, defslisp, etc. If your COMMON LISP environment requires some filetype other than .lisp, such as .lsp or .l, then rename the files accordingly. The tape also contains the text of this Users Guide and the Software Guide in \LaTeX\ format\(^3\).

The next step depends on what type of system you are on. If you are on a Symbolics system, then you will want to set up CL-Protos as a system so that you can then use the commands Compile System and Load System. Edit the file protos.translations to set the name of the physical host and directory, then move the files protos.system and protos.translations to the SYS:SITE directory. (There are equivalent system facilities

---

\(^1\)UNIX is a trademark of AT&T Company.

\(^2\)The device name /dev/rst0 is for Sun workstations; a different name may be required on other UNIX systems.

\(^3\)The \LaTeX\ text of this document is in files named *.tex and *.bib, and is provided so that you may create local versions of the Users Guide. To format the document, the following commands are used on a UNIX system: latex cl-protos, bibtex cl-protos, then latex cl-protos again, then print the results.
on other machines such as the TI Explorer — I just don’t know the details of how to set them up). Next, compile the source programs by entering the command:

```
Compile System protos
```

If your Lisp environment does not support a system facility, then you may use the simple one provided on the tape. Edit the file `load-protos.lisp`. At the top of the file are four constants specifying the name of the host, name of the directory, and suffixes for source files and binary (compiled) files. A couple of examples of these constants are shown below:

```
  Symbolics (defconstant *hostname* "PEARL:"
             (defconstant *dirname* ">dvorak>cl-protos>
             (defconstant *sctype* ".lisp"
             (defconstant *bintype* ".bin"

  UNIX    (defconstant *hostname* nil)
            (defconstant *dirname* "/usr/dvorak/cl-protos/"
            (defconstant *sctype* ".l"
            (defconstant *bintype* ".b"
```

Next, type:
```
  (load "load-protos.lisp")
```

and the following menu will appear:
```
Select an action by typing a single letter:
-----------------------------------------------
C   Compile the CL-Protos source files.
S   load the Source files for interpretive execution.
B   load the binary files for execution.
Q   Quit (do nothing).
```

Type "C" to compile the source files.

## 5.2 Loading

The preceding instructions are performed once by the person installing CL-Protos. The following instructions are aimed now at the people who will use CL-Protos. To load CL-Protos on a Symbolics system, type:

```
Load System protos
```

On other systems, type:
```
  (load "hostname dirname load-protos.lisp")
```

4The constant *hostname* may be set to nil if the files reside on the host you are running on. The files appear to be part of your local file system (even though they may reside on a remote file server).
where hostname and dirname are as specified in the load-protos.lisp file. For example, on a UNIX system you might type:

(load "/usr/dvorak/cl-protos/load-protos.lisp")

and, in response to the menu, type “B” to load the binary (compiled) version of CL-Protos. Loading may take a couple minutes, depending on the speed of the system.

5.3 Running CL-Protos

When loading is complete, type:

(in-package ’protos)
(protos)

CL-Protos will display a salutation message and prompt you for your name or initials (your name is stored with any knowledge that you add to the category network). Next, it will ask if it should keep a log of this session, to which you may reply “Y” or “N”. Logs are kept in a text file named protos-log, to be created in your current working directory. If it already exists, CL-Protos will append to it. Currently, the only information written to the log file is “explanation protocols,” i.e., English descriptions of explanations and their Protos-language counterparts. Other information may be added to the log file in the future.

From here on, CL-Protos is largely menu-driven and, hopefully, easy to explore and experiment with. For example, here is how the first menu will appear:

<table>
<thead>
<tr>
<th>TOP-LEVEL MENU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q go to Overview menu</td>
</tr>
<tr>
<td>D go to Display menu</td>
</tr>
<tr>
<td>C go to Control menu</td>
</tr>
<tr>
<td>U go to Unfocused menu</td>
</tr>
<tr>
<td>E go to case Entry menu</td>
</tr>
</tbody>
</table>

Select an item --->

To select a menu item, type the corresponding character (and on UNIX systems, type Return). For all menus where it makes sense to “pop back up” to the previous level, typing the letter “Q” will quit the current menu. On the top-level menu, “Q” will cause an exit from CL-Protos, returning control to the Lisp listener. Let’s begin by typing “O” for the Overview menu, as shown below:

What would you like an overview of?

---

5The CL-Protos menus and other prompts and displays are all designed for a dumb terminal interface since that is all that COMMON LISP currently supports. So, for the sake of portability, CL-Protos forgoes the use of pop-up menus, mouse selection, and graphics.
5.4. **SAMPLE DIALOGUE — ENTERING A NEW CASE**

---

G General concept behind Protos  
S Suggestions for getting started  
T Tips for a teacher of Protos  
E Explanation language  
N Node (term) syntax  
R Relation syntax  
K Quantifier syntax  
C Conditions syntax  
Q Quit this menu

Select an item ——>

Select and read the various overviews from this menu, then type "Q" to return to the top-level menu. Next, type "X" to load the example knowledge base. This small knowledge base contains two exemplars of the category **chairs**, named **chair1** and **chair2** (see [Bar88, pages 15–16]). These exemplars are described with features such as **seat**, **backrest**, **armrests**, **pedestal**, etc. To examine this knowledge base, type "D" to go to the display menu:

**DISPLAY MENU (select something to display)**

---

G Generalization hierarchy  
F Functional hierarchy  
K causal hierarchy  
P Part-to-whole hierarchy  
W Whole knowledge base  
X Show transformation definitions  
T names of all Terms  
D Details about a given name  
C exemplar-containing Categories  
I 1-step relations from x  
R Relationship between 2 terms  
Q Quit this menu

Select an item ——>

Go ahead and try some of these menu items to see the type of information stored in the category network. In particular, select menu item "T" to see the names of all the terms, then type "D" to see the details about terms like **chairs**, **chair1**, **chair2**, **seat** and **backrest**. For example, the details of **backrest** appear as:

**BACKREST** (a term)  
Remindings: chairs 0.50  
Feature of: (chair2 chair1)  
Relations: backrest enables (holds person), strength = 0.90  
Created by: dvorak

When done examining the sample knowledge base, type "Q" to return to the top-level menu.

---

5.4 Sample Dialogue — Entering a new case

Let's observe CL-Protos at its intended task of classifying a new case. With the **chairs** knowledge base loaded, we'll enter a new chair case. From the top-level menu type "E" to go to the case-entry menu:

---

*The terms category network and knowledge base are used interchangeably here. To be precise, a category network is the Protos-specific organization for a knowledge base.*
CASE-ENTRY MENU

E Enter case as a list of features
C Copy-and-edit a previous case
S get Sample case for Chairs KB
D Display a selected term
Q Quit, return to top-level

Select an item --->

For ease in demonstrating Protos, a pre-stored sample case can be retrieved by typing "S". Protos then reports:

Entering a new case consisting of these features:
 ---> seat
 ---> pedestal
 ---> rigid-material
 ---> swivel

This new case is named chair3. Since swivel is a new term, Protos checks with the user in case the term was misspelled or unintended:

SWIVEL does not currently exist:

C It's correct; Create it as a new term.
R Re-spell it.
S this is a Synonym for an existing name.
F Forget this name.
D Display existing names.

Select an item --->

Type "C" to indicate that the new term is correct. At this point, Protos will begin displaying some of its internal processing since many of the tracing switches are turned on, by default. Eventually, it will present a proposed match of the new case chair3 to the exemplar chair1:
5.4. SAMPLE DIALOGUE — ENTERING A NEW CASE

CASE: Chair3

<table>
<thead>
<tr>
<th>FEATURE</th>
<th>Sim. Feature</th>
<th>Eff. Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>seat</td>
<td>1.00 seat</td>
<td>+++</td>
</tr>
<tr>
<td>seat</td>
<td>0.72 armrests</td>
<td>+</td>
</tr>
<tr>
<td>seat</td>
<td>0.81 backrest</td>
<td>+</td>
</tr>
<tr>
<td>pedestal</td>
<td>1.00 pedestal</td>
<td>+</td>
</tr>
<tr>
<td>rigid-material</td>
<td>0.90 metal</td>
<td>+</td>
</tr>
<tr>
<td>spurious</td>
<td>1.00 wheels</td>
<td>-</td>
</tr>
</tbody>
</table>

Match Similarity = 0.90

This display contains a lot of information, so let's examine it in detail:

- The name of the case and its features appears on the left side while the name of the matched exemplar, its category, and its features appear on the right.

- A similarity value in the range 0–1 shows the strength of each feature-to-feature match, and the Match Similarity at the bottom shows the overall strength of the match.

- Feature-to-feature matches that are identical are emphasized by a row of equal signs (====) while matches that involve an explanation are shown with a row of dashes (-----). Matches that include a mutual exclusion relation are denoted by a row of exclamations (!!!!!).

- The importance of each exemplar feature is shown qualitatively at the right-hand edge. The values ++++, ++++, ++, +, and - correspond to strengths of necessary, high, moderate, low, and spurious, respectively.

- The match is always shown with the most important exemplar feature on line 1, proceeding down to the least important feature.

- Notice that the same case feature may appear more than once on the left side if it matches more than one exemplar feature. (Exemplar features appear only once).

- Exemplar features that are spurious (i.e., of no importance to the category) are flagged as such, as shown on line 6. Case features that are unmatched are also flagged, as shown on line 7. Such unmatched case features do not weaken the overall match, so no similarity value is shown for line 7. (Unmatched exemplar features are also flagged, and these do weaken the overall match).

After displaying a match, Protos asks:

Do you wish to examine any of the explanations? (Y or N)

If you reply "Y", then Protos prompts for an explanation number. In this example, if you type "3" Protos responds with:

As described later in the control menu, the "N" mode switch controls which of two similarity calculations is displayed.
CHAPTER 5. INSTALLING & RUNNING CL-PROTOS

3 SEAT enables (HOLDS PERSON), (HOLDS PERSON) is enabled by BACKREST

When you are done examining explanations, type "Q". Next, Protos asks:

Is CHAIRS the correct classification for this case? (Y or N)

The answer to this question determines the nature of the remaining dialogue. If the answer is "N", then Protos attempts to learn from the mistake before trying its next best hypothesis. In this example, however, we answer "Y". Protos then asks:

Is this a suitable exemplar for matching the new case? (Y or N)

If the answer is "N" then Protos will match the new case to the next best exemplar of the chairs category. If the answer is "Y" then Protos is almost done with this case, but first it gives the teacher the opportunity to enter any additional information, through the following menu:

UNFOCUSED INSTRUCTION MENU

===============================================================================
| T | enter new Terms/abbrev/synonyms | R | Remove a Relation |
| E | Enter new explanations           | X | define a transformation |
| C | add Censor                       | D | go to Display menu   |
| I | add/change an Importance         | Q | Quit this menu       |

Select an item --->

We'll decline by typing "Q". Protos then advises that:

This case CHAIR3 is similar enough to CHAIR1 that Protos is prepared to forget the case (and just retain CHAIR1). Will it ever be important to distinguish between these two? (Y or N)

If we answer "N", then Protos will summarize the fact that the new case was successfully matched to CHAIR1, and will then prompt for the next case.

Now, let's look at the situation where the teacher decides that a new case is not sufficiently similar to any of the hypothesized exemplars, and therefore wants to create a new exemplar from the case. To demonstrate this, we can again use the sample case (chair3). Starting at the case-entry menu, select "S" to get the sample case, then answer all the other questions as before until you reach the question:

Is this a suitable exemplar for matching the new case? (Y or N)

Type "N". Protos will then ask:

What would you like to do?
T Try the next most prototypical remaining exemplar.
S Select a specific exemplar of this category.
C Create new exemplar from this case.
A Abandon this case.
Type “C”. Protos then reports:

Case CHAIR3 is composed of the following features:
(seat, pedestal, rigid-material, swivel)

Are any of these features IRRELEVANT
to its role as an exemplar of CHAIRS? (Y or N)

Type “N”. Protos then reports:

Protos is currently unaware of the relevance
of 2 features to the category CHAIRS.

What is the relevance of RIGID-MATERIAL to CHAIRS?

-------------------------------
E Enter an explanation.
L let Protos Look for an explanation.
S declare the feature Spurious.
U leave this feature Unexplained.

Select an item --->

Type “L”. Protos will find an explanation that relates rigid-material with chairs. Next, it will ask:

What is the relevance of SWIVEL to CHAIRS?

-------------------------------
E Enter an explanation.
L let Protos Look for an explanation.
S declare the feature Spurious.
U leave this feature Unexplained.

Select an item --->

Since we know that swivel is a new term to Protos, there is no point in letting Protos look (unsuccessfully) for an explanation. Type “E”. Protos then says:

Please explain how SWIVEL is related to CHAIRS (preferred)
or how CHAIRS is related to SWIVEL (alternate form).

Please enter your thoughts first in English,
and terminate with a blank line.
(text):

When Protos asks for an explanation, it allows the teacher to first express his/her thoughts in free-form English text. In this case, a teacher might respond with:

A swivel is often part of a chair.

After entering a blank line to terminate the English explanation, Protos then says:
Now, translate this into one or more Protos-style explanations, enclosed in parentheses (terminate with blank line):

Here, we type:

(swivel usually part of chairs)

This is a simple 1-step explanation relating swivel to chairs. In situations where a multi-step explanation is appropriate, it is sufficient for the teacher to enter only the explanation steps that Protos doesn’t already know.

Protos is now ready to learn a reminding and an importance from this feature-to-category explanation:

Protos is ready to install a 0.56 reminding from SWIVEL to CHAIRS.
Should it instead install it to the more general category (HOLDS PERSON) ? (Y or N)

Type “N”. Having learned a reminding, Protos next tries to estimate an importance from the explanation:

Protos believes that SWIVEL is of MODERATE importance to CHAIRS
based on the explanation:
   CHAIRS usually has part SWIVEL

Do you agree?
  Y  Yes.
  E  no, let me revise the Explanation
  I  no, let me just revise the Importance

Select an item --->

Type “Y”. After a few more lines of output, Protos will attempt to learn a new “difference link”:

Protos previously mistook this case for the exemplar CHAIR1.
The features of this case that were not matched by CHAIR1 are:
   SWIVEL

Which of these features (if any) are important discriminators?
---> SWIVEL ?  (Y or N)

If you type “Y”, Protos will install a new difference link between chair1 and chair3. Protos has exhausted all of its questions and now summarizes the new exemplar that has been created:

The following new exemplar has been created:

CHAIR3 (an exemplar)
   Category:  chairs, typicality = 1.0
   Features:  (seat, pedestal, rigid-material, swivel)
   Created by:  whoever
This has been a simple example of a dialogue between CL-Protos and the teacher. There are several other queries that CL-Protos will generate for other situations; the best advice here is “explore and experiment!”

5.5 Escapes

Sometimes, when Protos is prompting you for an answer to some question, you may want to go check something in the knowledge base before answering. This can be done by temporarily escaping out of the query. To escape, type a question mark (?) followed by the desired escape code. If you don’t know the escape codes or have forgotten them, type two question marks (??). This will produce the escape menu:

```
ESCAPE OPTIONS

D go to Display menu.  ? Show these options.
C go to Control menu.  B Break (enter the debugger).
U go to Unfocused menu.  F show File names in current directory.
O go to Overview menu.  Q Quit (return to original prompt).
```

Enter option --->

For example, if Protos is waiting for a response and you type “?D”, the display menu will appear and you can display as many items as you wish. When you finally quit the display menu, the original query that you had escaped from will be redisplayed.

5.6 Control Menu

The control menu, which is selected from the top-level menu, controls several mode and trace switches that affect the operation of CL-Protos. The current value of each mode and trace switch is shown on the right side, with “T” corresponding to “on” and “NIL” corresponding to “off”.

```
CONTROL MENU  (select something to toggle)

Modes:

L Learning (vs. performance)  (T)
M Maximize learning  (T)
Z Apply heuristics  (T)
D Medical diagnosis model  (NIL)
X Critique explanations  (NIL)
W Nth root of similarity  (T)
A Abbreviated names  (NIL)

Tracing:

R Reminders and censors  (T)
```
CHAPTER 5. INSTALLING & RUNNING CL-PROTOS

I  Importances
H  Hypotheses
K  Knowledge-based pattern matching (NIL)
Y  KBPM relations traversed (NIL)
U  heuristics (inside kbps) (NIL)
C  Case matching
S  transformations & computations (T)
Q  Quit this menu

Select an item --->

The meaning of each menu item is elaborated below.

L  In learning mode Protos is allowed to interact with the teacher as it processes a new case. When this mode is NIL, CL-Protos runs in “performance” mode and performs classification without any teacher interaction.

M  When Protos is in learning mode, this mode means that Protos should maximize its opportunity for learning from the teacher (as opposed to trying to use its own knowledge).

Z  This mode switch controls whether or not the heuristic rules are applied by the knowledge-based pattern matcher. If this switch is NIL, Protos essentially performs a uniform-cost search without regard to the semantics of the relations among terms. This corresponds roughly to “free association.”

D  This switch controls the application of a medical diagnostic model in acquiring and processing knowledge. Currently all that the model does is ask about the medical significance of each new term; further work is needed before this becomes a useful mode.

X  When this mode switch is T, the teacher can critique explanations found by the knowledge-based pattern matcher. The explanation and the critique are written to the log file for later study in improving Protos’ generation of explanations.

N  This mode affects the calculation of overall similarity in a new-case-to-exemplar match. If NIL, similarity is calculated as simply the product of the strengths of each feature-to-feature match. If T, one additional step is taken — if there are \( N \) exemplar features, then we compute the \( N \)th root of the product. The effect is that the overall similarity of a match becomes the average strength of a feature-to-feature match.

A  When this mode is T, then abbreviations are used when printing the names of any terms, quantifiers, and relations. For example, the explanation “(fire always causes heat)” would be displayed as “(fire al cs heat)”. If abbreviations existed for the terms fire and/or heat, then these too would be used. Abbreviations can always be used during input, regardless of the setting of this mode. (Note: to create an abbreviation for a term, go to the Unfocused menu).

R  Traces the creation of and processing of reminders and censors.
5.7. **Transformations**

CL-Protos provides a mechanism for transforming case features from their given form to a form more suitable for classification. For example, you may define a quantitative-to-qualitative transformation to convert *(temperature 100)* to *(fever mild)*. You may also define a computational transformation such as for converting *(length 4)* and *(width 3)* to *(area 12)*. Transformations can be defined from the Unfocused Instruction Menu and displayed from the Display Menu. When defining a transformation, CL-Protos puts you in the editor with a skeleton Lisp function to be edited. Transformations are an experimental feature (and have not been thoroughly tested).

5.8 **KB Files**

At the top-level menu, CL-Protos provides the capability of saving a knowledge base (KB) to a file and loading a new knowledge base from a file. These “KB files” are text files and have the same format across all systems. In addition to the obvious benefit of saving one’s work on non-volatile storage, it also allows knowledge bases to be exchanged among users of CL-Protos and used on different types of systems.

A KB file has a simple format that is intended to be easy to read and understand. For example, here is an excerpt from the KB file representing the chairs KB:

---

8 You can create the KB file for the chairs knowledge base by typing X at the top-level menu to load the chairs KB and then typing S to save it to a file. “X” does not load the chairs KB from a KB file; rather, it runs the load-example function which generates the KB. This is a quirk of history since the load-example function was created and used long before the KB load/save feature was ready. Although we could remove the load-example function and distribute the chairs KB as a KB file, it has often been useful to have a “built-in” example for development, testing, and demonstration.
A KB file can be edited with a text editor. This may be useful in some situations since CL-Protos doesn’t provide every imaginable editing operation, such as renaming a term or adding a comment to an existing exemplar. After editing a KB file, make sure that all parentheses are balanced. Also, if you delete a relation, be sure to also delete the inverse relation. Otherwise, CL-Protos will re-create the relation when it reads and installs the inverse relation.

5.9 Terminal Input Idiosyncrasies

There are some differences in the terminal input characteristics of Lisp machines, UNIX systems, and HP’s NMODE that affect the way the teacher enters input to CL-Protos:

Lisp machines: Lisp machines such as Symbolics and TI Explorer know when to complete a read operation based on the type of read (read, read-char, or read-line) and the actual input. For example, read-char completes with any keystroke, whereas read completes upon seeing any white-space character following an atom or the closing parenthesis of the outermost list. In effect, the various read operations on a Lisp machine complete at the earliest possible instant.

UNIX systems: On a UNIX system, Common Lisp runs as an application program and uses the underlying UNIX I/O system calls. Unless such an application program takes special measures (which Kyoto Common Lisp apparently does not do), the terminal input will be in “cooked” mode (as opposed to “raw” mode) and therefore every type of read will have to be completed by typing Return. This means, for example, that selecting a menu item (which was intended to be done with a single keystroke) requires two keystrokes on a UNIX system. (On the other hand, it has the virtue of consistency in that every read operation requires a Return keystroke on UNIX).

HP’s NMODE: Although the HP 9000 machines are UNIX systems, they run Common Lisp in NMODE, a program that integrates EMACS editing, Lisp execution, and
output history capture. In this environment, terminal input to a read operation is completed only when the Enter key is depressed (not the Return key). Furthermore, the input seen by the read operation begins with the first character on the line, which is not necessarily the first character typed. For example, when CL-Protos prompts with Select an item --->, leaving the cursor at the end of the prompt, then if the user types some character followed by Enter, the read-char will return “S” because that was the first character on the line. One solution to this problem is for the user always to type Return to begin a new line, then type the desired input followed by Enter. The other solution is to establish a separate window pane in NMODE just for responding to prompts. Specifically, you see the prompt in the “output” window pane but enter all responses in the “input” window pane.

5.10 Miscellaneous

- If you ever abort out of CL-Protos (returning to the Lisp listener), you won’t lose the current category network. Simply reinvoke CL-Protos by typing “(protos)” and you’ll find the category network intact. CL-Protos does not do any initialization that affects the category network.
- The “?” escapes, in theory, should be available at every prompt, but it probably hasn’t been inserted everywhere yet.
Part II

CL-Protos Software Guide
Chapter 6

Introduction

This software guide is intended for those who wish to study and/or modify the CL-Protos program. The source code is fairly long (about 13,000 lines), so this guide attempts to serve as a map of the territory, pointing out the highlights but leaving the details for personal investigation. The guide presents the high-level design of CL-Protos, describing the top-level algorithm and the major data structures and functions.

6.1 Related Documents

CL-Protos is a reconstruction of the Protos exemplar-based learning apprentice based primarily on the written description of the Protos paradigm in [Bar88, chapter 3]; that document is highly recommended as a companion to this software guide. Also, Part I should be read first for an understanding of the external interface presented by CL-Protos.

If you are going to modify or extend some part of CL-Protos, here's what I recommend. First, read the relevant material in [Bar88, chapter 3]. Then, consult this Software Guide to help you determine the relevant structures and functions in CL-Protos. Finally, study the appropriate portions of the source code. We have tried to make the source code easily understandable by providing extensive source code comments on the purpose of each major structure and function.

6.2 Design

In terms of design, CL-Protos is a mixture of procedural and object-oriented programming. It is object-oriented only to the extent that it defines structures for the major objects of the category network and for returning structured results from certain functions. For example, there are defstructs for cases, features, categories, exemplars, difference links, relational links, verbs, quantifiers, explanations, feature-to-feature match results, and case-to-exemplar match results. The rest of the code is procedural in style.

CL-Protos was designed to be a research tool rather than a production system. Many
improvements can doubtless be made to improve efficiency, human interface, packaging, and documentation.

6.3 CL-Protos Source Code

CL-Protos consists of about 13,000 lines of source code in 17 files, developed initially on Symbolics Genera 7.2. The program should be completely portable to any Common Lisp environment since it adheres to the language description in Common Lisp: The Language, by Guy L. Steele Jr. (it does not use any extensions offered in Symbolics Common Lisp such as New Flavors, nor does it use the proposed Common Lisp Object System). To date, CL-Protos has run on Symbolics Genera 7.2, TI Explorer (Release 3.1), HP 9000 Series 300, Sun 3 (Kyoto Common Lisp), and Macintosh (Allegro Common Lisp). If you encounter any problems in running CL-Protos on a different system, please let us know. Previous experience has shown that there are a few differing interpretations of the language definition, mostly dealing with input/output, but so far we have been able to find a “least common denominator” that works on all systems.

6.4 Source Code Conventions

- Preceding each structure there is a comment block describing the role of the structure and its relation (if any) to other structures.
- Preceding each major function there is a comment block detailing the function’s arguments and its purpose. In some complex functions, the overall design is described.
- Comments follow the conventions described in [Ste84, page 347] regarding the use of semicolons (; ; ; ; and ; ; ;).
- Global constants and variables have names that begin and end with an asterisk, e.g., *reminding-moderate*.
Chapter 7

Global Objects

CL-Protos defines a number of global constants and parameters, most of which are in the files `global.lisp` and `verb.lisp`. CL-Protos also creates global objects as the category network is built.

7.1 Numeric Constants

CL-Protos defines a number of constants whose values are somewhat arbitrary, such as the strength value of a “moderate” reminding. The intent has been to collect all such constants into `global.lisp` where they can be seen and experimented with. One other file, `verb.lisp`, contains numeric constants for the strength values of the various relations and quantifiers. It is believed that CL-Protos’ performance is not overly sensitive to modest changes in these constants — the main requirement is that they be reasonably consistent as a group. For example, the strength of a moderate reminding should be more than a weak reminding but less than a strong reminding. Likewise, the strength of a strong relation such as “causes” should be comparable to the strength of a strong reminding. Briefly, the numeric constants are these:

- Lower bounds on five different qualitative strength of remindings (absolute, strong, moderate, weak, and very-weak);

- Lower bounds on five different qualitative strengths of importances (absolute, necessary, high, moderate, low);

- Threshold values for exemplar-to-new case matches (strong match, weak match, minimum acceptable match, and “near miss” threshold);

- Strength of each relation and quantifier (in `verb.lisp`); and

- Various limit values, such as the maximum number of exemplars of a single category that may be put onto the hypothesis list.
7.2 Switches

All of the trace and mode options that appear in the control menu are defined in global.lisp. Each switch is either T or NIL.

7.3 Anchors

CL-Protos maintains three global lists for the purpose of being able to save its current knowledge and/or forget its current knowledge. The global parameter *history* contains a list, each element of which is a term, a predicate, or a case, roughly in the order in which they were first mentioned by the teacher (or first appeared in a KB file). As new terms, predicates, and cases are created, they are added to this list. The list is read only when the teacher has said to save the knowledge base to a file or to forget the current knowledge base. The other two list anchors are *transformations* and *uninstantiate-predicates*, both of which are used in transforming input features (such as a quantitative-to-qualitative transformation).

7.4 Menus

All of the user-interface menus are global objects, typically defined near the function(s) where they are used. The file protos.lisp contains the prominent, high-level menus — the top-level menu, overview menu, display menu, control menu, unfocused instruction menu, and case-entry menu. The file discuss.lisp contains most of the other menus used in the discussion of a new case. A few other menus are scattered throughout the other files.

7.5 Verbs & Quantifiers

The file verb.lisp contains a global parameter for each verb and each quantifier of the explanation language. These parameters, such as *verb-isImpliedBy* and *quant-usually*, are required mainly by the explanation parser (parse.lisp). Also, a number of functions that look for specific "hierarchical" relations in the category network (e.g., generalization, functional, causal, or part-to-whole relations) test for specific verbs using these global parameters.

7.6 Category Network

Every propositional term, predicate name, and propositional argument name in CL-Protos becomes a bound symbol in the protos package. For example, if a teacher enters the explanation:

\[(\text{infection usually causes (fever mild)})\]
then the symbols infection, fever, and mild will become bound (if they weren't already bound). Thus, every time that CL-Protos encounters a name in teacher-provided input, it checks to see if the name is bound. If so, and if the evaluated name is of type node, then it represents a known object in the category network. If not bound, then the name is new and becomes bound to a newly-created object of the appropriate type (a term or predicate).¹

¹The fact that names in the category network are bound symbols in the protos package lets CL exploit the ability of the eval function to convert a name to a storage location.
Chapter 8

Structures

Most of the structures used in CL-Protos are defined in the file `defs.lisp`, and there are many comments in this file to aid in understanding why each structure exists and what its contents are. Thus, it's best to examine `defs.lisp` while reading this chapter. The structures described below are arranged in related groups rather than alphabetically.

8.1 Terms

This first group of structures are all closely related in that their slots are all included in instances of a `term`. In Protos terminology, a `term` is the combination of a feature, a category, and an exemplar. In CL-Protos, a `term` is a "super structure" that includes the slots of the `feature`, `category`, and `exemplar` structures. This super structure is necessary because, regardless of whether a term begins life as a feature, a category, or an exemplar, it may eventually be used as one of the others. For example, the term `engine` might be a feature of the case `my-car` and at the same time might also be a category with specializations such as `piston-engine` and `Wankel-engine`. Thus, CL-Protos `never` does a `make-feature`, `make-category`, or `make-exemplar` — it does a `make-term` instead.

**term**  A `term` is created for every named object of the category network. Although a term includes all the slots of `feature`, `category`, and `exemplar`, it also contains two slots of its own. If the term's name is in the form of a predicate with arguments (`e.g., (fever mild)`), then one slot points to the appropriate `predicate` structure (`i.e., the fever predicate`). The other slot contains the "type" of the term; this is used only in the medical diagnosis mode to distinguish among observations, laboratory findings, diagnoses, etc.

category  This structure contains the slots specific to a category, namely, a list of exemplars of the category and an association list of features and their numerical importance to the category.
8.2. PREDICATES

**feature** This structure contains the slots specific to a feature, namely, an a-list of remindings (pairs of (category . strength). There is also a list of the exemplars of which this feature is a feature; this slot is used only in handling “featural exemplars” and may be safely ignored if CL-Protos is not classifying multi-level concepts.

**exemplar** This structure contains the slots specific to an exemplar, namely, the category of the exemplar, the list of features of the exemplar, the typicality of the exemplar within its category, and a list of difference links (if any).

**node** This structure defines slots that are needed by other named objects of the category network. Mainly, these slots are for the object’s name, abbreviation (if any), synonyms (if any), list of relational links, and the name of the person who created this object. Nodes are created only as a result of make-term or make-predicate.

8.2 Predicates

**predicate** Terms in CL-Protos may be expressed either in the form of a proposition (e.g., infection) or a predicate with arguments (e.g., (fever mild)). For the former, the proposition name is used as a Lisp symbol whose value is set to the instance of its term structure, i.e., (setq infection (make-term ...)). However, since list forms cannot be used as symbol names, then predicate-with-arguments terms must be handled differently. The predicate structure provides a directory that maps a specific combination of arguments of a given predicate to its associated instance of the term structure. More detailed comments about this are in the source code.

8.3 Relational Links

**relation** This is the structure for relational links in the category network. Consider the relation (if category is cars then rubber usually is part of tires). This relation has a condition (if category is cars), a quantifier (usually), a relational verb phrase (is part of), and an antecedent and consequent (rubber and tires). Slots for all of these components exist in the relation structure. In addition, there is a slot for the inverse relation from tires to rubber.

**verb** There is an instance of the verb structure defined in verb.lisp for every verb in the current CL-Protos explanation language. Each verb has a name, an abbreviation, a strength, an inverse quantifier, and two flags used by the knowledge-based pattern matcher indicating how many antecedents and how many consequents need to be “proved” (if there are multiple antecedents or consequents). This depends on whether the verb is active (e.g., causes), pas-

---

1 a-list ≡ association-list
sive (e.g., is caused by), bidirectional (e.g., cooccurs with), or non-directional (e.g., has typical generalization).

**quantifier** There is an instance of the quantifier structure defined in verb.lisp for every quantifier in the current CL-Protos explanation language. Each quantifier has a name, an abbreviation, and a strength.

**condition** An instance of this structure represents a condition which may be attached to one or more relations. Slots exist for the type of condition (e.g., if category is X; if exemplar has features (A and B); if newcase has features (A and B)), and for the data required for each condition type (i.e., a category or a list of features).

### 8.4 Difference Links

**difference** This is the structure for difference links in the category network; such links are attached between pairs of exemplars. More precisely, a difference link from A to B is stored in the exemplar-differences slot of A and the link contains a slot for the target exemplar (B) plus a list of features that distinguish B from A.

### 8.5 Explanations & Matches

**explanation** Explanations are generated as the result of a successful (but not identical) match found by the knowledge-base pattern matcher from a search through the category network. While a relation contains all the information about a single inference step, an explanation can represent a chain of inference steps or, more generally, tree of inference steps. An explanation is an inherently hierarchical structure. The 1-step explanation A causes B can be stored in a single explanation structure, but the 2-step explanation A causes B implies C requires two levels of explanation, namely, [A causes [B implies C]]. Each explanation has slots for a relation, a strength, antecedent terms, consequent terms, and a “start term”. The antecedent and consequent terms will often be identical to their counterparts in the associated relation, but they may be a subset if the relation had multiple terms and the verb was such that only one of the terms needed to be proved for the explanation to hold.

**result** A result instance is generated only by the knowledge-based pattern matcher to return the result of an attempted feature-to-feature or feature-to-category match. Slots exist for the feature being matched, its importance, the type of result (identical match, explained match, spurious feature, mutual exclusion in match, or unmatched), and the associated explanation, if any.

**match** A match instance is the next level up from a result, and represents an exemplar-to-new-case match. It contains slots for the exemplar, the new
8.6 **KBPM Search Graph**

The following two structures are used only by the knowledge-based pattern matcher for holding the intermediate state of the uniform-cost search. These structures are defined in *kbpm.lisp* and used there as well as in *heuristics.lisp*.

- **graphnode**: An instance of this structure is created once for each term the first time it is encountered in a uniform-cost search. It contains slots for the associated **term**, the strength of the path from the start node to this graphnode, the parent graphnode (if any), and several other slots necessary to the uniform-cost search.

- **graphlink**: This structure links graphnodes in the search graph. This structure is somewhat similar to **explanation** in that it contains slots for the relation and the antecedent and consequent nodes actually used in the search. When a search is successful, the appropriate explanation is extracted from the successful path in the search graph.

8.7 **Other**

- **case**: A **case** is created for each new case presented to CL-Protos. It contains slots for the name of the case, its features, its category (if it was pre-classified), and its final disposition. In the course of its short life, a case either becomes a new exemplar or is "merged" with an existing exemplar or is abandoned. Although a case is never accessed after this, it is in fact saved on the *history* list as part of the knowledge base and could be used in the future if code were added to CL-Protos to do something with these "old" cases.

- **transformation**: An instance of this structure is attached to each feature that requires "transformation" before use by CL-Protos. Most typically, these are quantitative-to-qualitative transformations, as in transforming (temperature 98.6) to (temperature normal).

- **menu**: This structure is for defining menus for the human interface of CL-Protos. Most of these menus are static; several examples can be found in the file *protos.lisp*, although others are scattered throughout the software wherever a multiple-choice must be presented to the user. Slots exist for the list of items in the menu as well as for options that affect the behavior of the menu, *e.g.*, 1-column vs. 2-column mode and single vs. multiple selections.
8.8 Non-Structures

Most of the structures described above are named after items described in [Bar88]. However, some items do not have associated structures — reminders, censors, importances, hypotheses, similarity, and category network\(^2\). This section describes how these items are represented and where they are stored.

**reminders** Inside a category network, a reminding from the feature \(F\) to the category \(C\) is stored as the pair \((C, \text{strength})\) in the association list stored in the **reminders** slot of feature \(F\). In other words, \((\text{feature-reminders } F)\) contains an a-list of all reminders associated with feature \(F\).

**censors** Censors are stored exactly as reminders — the only difference is that their strength values are negative.

**importances** Inside a category network, the importance of feature \(F\) to category \(C\) is stored as the pair \((F, \text{strength})\) in the **importances** slot of category \(C\). In other words, \((\text{category-importances } C)\) contains an a-list of all featural importances known for category \(C\).

**hypotheses** Hypotheses are temporary objects that exist only while a new case is being classified. Hypotheses are created by the function **build-hypotheses**. Each hypothesis is a 3-element list:

\[
(\text{exemplar strength lastp})
\]

where **exemplar** is the hypothesized exemplar to be matched with the new case, **strength** is the strength of the combined reminding to this exemplar, and **lastp** is non-nil if this is the last hypothesis involving this exemplar's category.

**similarity** "Similarity" is the end result of an exemplar-to-new-case match — the single numeric value produced by the Protos similarity metric. It is stored in the **similarity** slot of a **match** structure. Thus, a given similarity value exists only as long as its associated **match** exists.

**category network** A category network is the name for the body of knowledge contained in the **terms** and the links among them (relational links and difference links). CL-Protos maintains a single category network; there can be separate, unconnected networks within the category network representing unrelated chunks of knowledge.

---

\(^2\) Previous publications by Bareiss and Porter use the phrase **category structure**, but within CL-Protos we use the phrase **category network** to avoid confusion with Common Lisp's structures.
Chapter 9

Functions

9.1 Protos Classification Algorithm

The following exhibit, taken from [Bar88, page 18], is a high-level view of the Protos classification algorithm. There are several more detailed algorithm exhibits in [Bar88] that should be consulted as you get into the details of Protos.

Input a case to be classified.
Retrieve reminders based on the features of the case
and heuristically combine them {hypothesis formation}.
REPEAT
  REPEAT
    Use the strongest combined reminding to select an
    exemplar from the category structure.
    Evaluate the similarity of the new case to the
    exemplar by constructing an explanation of their
    similarity {hypothesis confirmation}.
    UNTIL an adequately explained match is found.
    Use difference links to improve the classification by
    considering neighboring exemplars as alternatives.
    Present the match and explanation to the teacher.
    IF the teacher disapproves
      THEN {Learn from the failure}
        Request new domain knowledge from the teacher.
        Reassess the reminders which suggested the exemplar.
      UNTIL the teacher approves.

9.2 Who Calls Who

This section presents a top-down view of CL-Protos as a set of pseudo-code exhibits showing the major functions and who they call. The notation used is demonstrated in the following example.
alpha
beta
[gamma]
[delta]
epsilon

This exhibit means that the function alpha calls beta, and then beta might call gamma and might call delta. Upon return from beta, alpha then calls epsilon.

The following pseudo-code exhibits are meant to focus attention on typical calling patterns involving major functions. Not every function call that actually appears in the source code will appear in these exhibits. This is provided to help orient you around the many CL-Protos functions. You should examine the actual source code for a more detailed understanding.

9.2.1 protos

The top-level function of CL-Protos is protos and is the only function which most users will ever call directly. Protos calls menu-select to display the top-level menu. In a typical situation where the user opts for the case-entry menu, the function enter-case-and-classify is called which prompts the user for the new case and then classifies the case and displays the results.

protos
  menu-select *top-menu*
  enter-case-and-classify
  enter-new-case
  classify-and-display
    classify
    display-classification
  [save-kb]

9.2.2 classify

The classify function is the top-level "program-callable" interface to the Protos classification procedure. Basically, if you plan to use the Protos classification procedure as a utility for another program, this is the function to call. Classify takes one argument, the new case, and returns a list of matches sorted in decreasing order of similarity to the new case. This is the behavior if the global parameter *learning-mode* is nil. Otherwise, it will return the one match approved by the teacher.

classify
  get-raw-reminders
  combine-reminders
  build-hypotheses
9.2. WHO CALLS WHO

test-hypotheses

9.2.3 get-raw-remindings

This function collects the remindings associated with each feature of the new case. It returns an association list with each entry of the form (reminded-category . strength).

9.2.4 combine-remindings

The raw remindings evoked by the features of a new case undergo considerable processing in this function before a set of hypotheses are generated. A general theme in much of this processing is to favor the most specific classifications. This is done by allowing reminded exemplar-containing categories to inherit the strength of reminded superordinate categories, by generating remindings for categories that are specializations of two or more reminded categories, by allowing reminded exemplars to inherit the strength of its category’s reminding, and by allowing a reminded term to inherit any reminding to a term that is superior in the sense of “is exemplar of” or “is feature of” (this last technique only affects featural exemplars). The end result of combine-remindings is a list of remindings sorted in decreasing order of strength, with the most specific terms having the highest strengths.

combine-remindings
merge-remindings
strengthen-categories
common-specialization
strengthen-exemplars
strengthen-features
sort

9.2.5 build-hypotheses

The build-hypotheses function takes as input a list of combined remindings and returns a list of hypotheses sorted in decreasing order of strength. For each reminded category, this function selects up to *n-best-exemplars* of that category and sets the strength of each hypothesis by multiplying the strength of the combined reminding to the category times the relative prototypicality of the exemplar within its category.

The subordinate function build-hypotheses2 is called only in the situation where there are no remindings to exemplars or exemplar-containing categories. In this case, we take as hypotheses the most prototypical exemplar of each of the subordinate exemplar-containing categories (subordinate in the sense of “has typical specialization”, “causes”, or “is full of”).

build-hypotheses
9.2.6 test-hypotheses

This function controls the major internal loop of the Protos classification algorithm — the testing of each hypothesis by matching it against the new case. If a match is poor (*minimum-match*) or is disallowed by a mutual exclusion, the hypothesis is rejected and the next hypothesis is tried. If a match is reasonably strong (*difference-threshold*) then Protos attempts to improve it by exploring the exemplar's difference links.

The function returns two values — an action (either 'done or 'retry) and a list of matches (possibly nil). The number of hypotheses tested depends on whether CL-Protos is in maximum learning mode, minimum learning mode, or performance mode. In maximum learning mode, CL-Protos performs one match at a time and presents each match to the teacher for discussion, stopping as soon as the teacher approves a match. In the other two modes, CL-Protos performs up to *n-best-matches* and then sorts the matches in decreasing order of similarity before either returning (in performance mode) or presenting the matches to the teacher (in minimum learning mode).

```lisp
(test-hypotheses
  (test-hypotheses2
    (compare-new-case
      (get-sorted-importances
        (compare-feature
          (kbpm
            [disallow]
            (explore-differences
              (compare-new-case
                [sort]
                (discuss-match
                  [add-new-exemplar]

9.2.7 discuss-match

The discuss-match function is the top-level entry to a large body of code devoted to discussion between CL-Protos and the teacher. All of the discuss-... functions are characterized by asking questions of the teacher and then branching to various other sections of code depending on the answer. While it is straightforward to explore a particular path through this code, the code is relatively long and can descend several levels deep into subordinate functions.

Discuss-match is called when CL-Protos has just presented an exemplar-to-new case match and wants to discuss it with the teacher. The first (and only) question that discuss-match
9.2. **WHO CALLS WHO**

asks the teacher is "Is X the correct classification for this case?" If the teacher replies "yes" then it calls `discuss-success`, otherwise it calls `discuss-failure`. These two functions are responsible for conducting the dialogue with the teacher to learn from problem-solving successes and failures, respectively.

```plaintext
discuss-match
    [discuss-success]
    [discuss-failure]
```

### 9.2.8 `discuss-success`

This function is called by `discuss-match` when the teacher agrees that the matched category is correct. Given a match structure it returns: (1) the same match if the teacher agrees that the chosen exemplar is appropriate; (2) a different match if the teacher chooses a different exemplar; (3) or nil if no more action is needed in this match (such as when the teacher decides to make a new exemplar from the case, or the teacher decides to abandon the case). The algorithm is described briefly below.

1. Call `discuss-exemplars` to let the teacher select the appropriate exemplar within the approved category.

2. Call `discuss-unmatched` to ask the teacher for explanations linking the unmatched features of the case to the features of the selected exemplar.

3. Call `discuss-unfocused` to allow the teacher to enter any new relations into the category network.

4. If the new case is mergeable with the selected exemplar, the teacher is asked for permission to do the merging.

5. Otherwise, the teacher is given the option to abandon the case or to create an exemplar from the new case.

6. Call `update-prototypicality` to update the prototypicality of the matched exemplar.

```plaintext
discuss-success
    discuss-exemplars
    discuss-unmatched
    discuss-unfocused
    mergeable
    update-prototypicality
    [merge-exemplar]
    [add-new-exemplar]
```
9.2.9 discuss-failure

This function is called by discuss-match when the teacher says that the matched category is incorrect. The objective then is to learn as much as possible from the mistake before proceeding with other hypotheses. The algorithm is described below.

1. If the given incorrect match is strong enough, record it for possibly installing difference-links later.

2. Reassess the reminders that led to the wrong category classification.

3. Ask the teacher for censors (i.e., negative reminders).

4. Reassess the importance of low-importance unmatched features.

5. As a last opportunity for learning, allow the teacher to enter "unfocused instruction" (typically, new explanations).

6. Let the teacher decide what to do next — create a new exemplar from this case, retry finding a match for the case, rematch the case to the same exemplar, try the next hypothesis, or abandon the case.

   discuss-failure
   reassess-reminders
   discuss-censors
   discuss-unmatched-importances
   discuss-unfocused
   menu-select

9.2.10 add-new-exemplar

This function takes the current case, creates an exemplar from it, and installs it in the category network. A considerable amount of discussion accompanies the creation of a new exemplar. Specifically, CL-Protos:

- asks for the exemplar's category, if unknown;
- removes any features that the teacher says is irrelevant to its role as an exemplar of this category;
- asks the teacher for an explanation of the relevance of each feature to the category (for those features that Protos doesn't already know about);
- asks the teacher about lowering the importance of any high-importance features of an existing prototype of this category that cannot be matched to some feature of the new exemplar; and
- asks the teacher to identify which features discriminate between the new exemplar and any "near misses" that occurred while trying to match the new case to existing exemplars (difference links).
add-new-exemplar
   [read-category]
discuss-removals
make-term
discuss-relevances
   get-unknown-features
menu-select
   [search-explanation]
kbpm
discuss-explanation
   [discuss-explanation]
   [ask-for-explanation]
enter-explanation
discuss-explanation
   [set-spurious]
discuss-high-importances
discuss-near-misses

9.2.11 discuss-explanation

This is the function in which CL-Protos learns remindings and importances from an explanation, whether the explanation was supplied by the teacher or found in the category network. If it is a feature-to-category explanation, then the reminding strength is taken from the strength of that explanation since a reminding estimates \( p(C|f) \), the probability of the classification given the category. Then, the explanation is inverted into a category-to-feature explanation and the importance of the feature to the category is taken from strength of the inverted explanation since an importance estimates \( p(f|C) \), the probability of the feature's presence given the classification. If the given explanation is a category-to-feature explanation, then the importance is taken directly from that explanation and the reminding is taken from the inverted explanation.

discuss-explanation
   get-leaves
set-reminding
invert-explanation
set-importance

An inspection of the source code will show that set-reminding will truncate the explanation from which the reminding is taken if certain conditions are true, such as the existence of a \texttt{has-typical-specialization} relation in the explanation. Also, set-importance will always ask the teacher if the proposed importance value looks reasonable. The reason for this is that the strength of an inverted explanation may differ from what the teacher wants as an importance value.
9.2.12 compare-new-case

This function compares a single exemplar to the new case and returns the results in a match structure. Beginning with the most important non-spurious feature of the exemplar, the knowledge-based pattern matcher (kbp) is called in an attempt to find an identical or explained match from the exemplar feature to any feature of the new case. When the last exemplar feature has been processed, two similarity values are computed and stored in the match structure — the "normal" similarity described in [Bar88] (the product of the strengths of each feature-to-feature match), and the N-th root of that similarity (where N is the number of exemplar features). This "N-th root of similarity" is effectively the average value of a feature-to-feature match.

```plaintext
compare-new-case
  get-sorted-importances
  make-match
  for each feature do
    compare-feature
      kbp
  return match
```

9.2.13 explore-differences

This function is called by test-hypotheses2. Given the current exemplar-to-new-case match and a "closed" list of exemplars that have already been explored, this function returns the best match found, possibly the original match, or a better match found by recursively exploring the difference links emanating from the exemplar. The algorithm is shown below.

```
 For all difference links from exemplar do
   If difference link has features matching newcase
     then calculate total importance of matched features
     and add target exemplar to list of candidates.
 While list of candidates is not empty
   Match newcase to candidate.
   If similarity > current similarity
     then recursively improve match to candidate.
 Return current best match.
```

9.2.14 get-explanation

This function is the main parsing function for user-entered explanations. Given a properly formed "sentence" representing an explanation, such as (fire causes heat), this function installs the corresponding relations in the category network and returns an explanation structure. This function is called by enter-explanation. The algorithm is described below:
9.2. WHO CALLS WHO

Get the condition (if any) by calling get-condition.
Get the antecedents.
Get the quantifiers (if any) by calling get-quantifiers.
Get the relation's verb (or verb phrase).
Get the consequents.
Install the relation by calling install-relation.
If another step to this explanation, process it recursively.

9.2.15 install-relation

This function is called by get-explanation. Given the from-nodes, to-nodes, quantifiers (if any), verb, and condition (if any) of the relation to be installed, this function installs the relation in the category network. This is done by creating a relation structure and adding it to the list of relations associated with the antecedent node; and by installing the inverse relation in the consequent node. The algorithm is described below.

1. Adjust the strength of the forward relation by multiplying the verb strength and the quantifiers strength (if any).
2. Build an inverse quantifier list and adjust the strength of the inverse relation.
3. Create the relation structures for the forward and inverse relations.
4. Check if the new relation conflicts with an existing relation in the category network.
   If a conflict exists call the function install-relation2 to use heuristics or query the teacher to determine which relation is to be retained.

9.2.16 kbpm

This is the top-level entry to the knowledge-based pattern matcher. This function is called whenever Protos needs to search for a feature-to-feature match (as when it is matching an exemplar to the new case) or a feature-to-category match (as when Protos is trying to learn a reminding). Kbpm attempts to find a chain of relations linking a given feature to another feature, exemplar, or category. The function returns a result structure indicating the type of match (identical, explained, spurious, excluded, or unmatched), the strength of the match, and the explanation (if any). Since the knowledge-based pattern matcher is one of the largest components of Protos, we devote a fair amount of space here to describing it, particularly for those who may wish to modify it in the future.

The kbpm function views the category network as an AND-OR graph and performs a uniform-cost search on it. The search always begins from a feature (the exemplar feature in a feature-to-feature match), proceeding toward a goal which may be either a case feature, a category, or an exemplar. To keep track of the search in progress, kbpm maintains a search graph built up from instances of graphnode and graphlink structures. At any moment, each graphnode is on one of the global lists *open*, *closed*, or *limbo*. A graphnode typically begins on *open* and then moves to *closed*; when a graphnode must revisit the solution of other graphnodes, it begins on *limbo*. When the goal node is finally
found, kbpm extracts from the search graph the strongest path from start node to goal node, returning that as the explanation.

Parameters given to kbpm are:

- **Mode**: whose value is either 'FtoF (feature-to-feature search), or 'FtoT (feature-to-target, meaning feature-to-category or feature-to-exemplar search).

- **Feature**: The feature which is the starting point of the search.

- **Importance**: The feature’s importance, used to control the amount of effort to be expended while attempting a match.

- **Object**: The object of the category network that is being matched. This object is an exemplar when the mode is 'FtoF, or a category or exemplar, when the mode is 'FtoT.

- **Case-features**: the features of the new case (if this is a feature-to-feature match, otherwise nil).

The components of the search graph are described below:

- **Graphnode**:
  This structure serves as a node for the search graph that is constructed by the kbpm algorithm, and also serves as a storage site for all the necessary information needed to conduct the search. The graphnode structure contains the following slots: (1) name, pointer to the node in the category network; (2) downstrength, the strength of the traversed path from the start graphnode to the current graphnode; (3) certainty, the strength of the path propagated upwards from the goal graphnode to the current graphnode; (4) parent (if any); (5) status ('solved, 'spurious, 'failed, 'unknown); (6) tree (number of search tree); (7) num-relations, number of relations emanating from this node; (8) forward, true if the direction of search from this node is forward (false otherwise); (9) limbo, list of nodes which depend on this node being solved and thus, are on *limbo*; (10) marked, true if this node is in the solution path; (11) link, graphlink from parent to this node; (12) best-child-link, best outgoing link.

- **Graphlink**:
  This is the structure that links graphnodes within the search graph created by kbpm. The graphlink structure contains the following slots: (1) is-and, true for “and” relations; (2) num-siblings, number of sibling nodes within an and-relation; (3) parent.

- **open**:
  List of open graphnodes.

- **closed**:
  List of closed graphnodes.

- **limbo**:
  List of graphnodes dependent on other graphnodes to be solved.
The basic calling hierarchy of kbpm is shown below, followed by a brief description of its main functions.

```
kbpm
  uniform-cost-search
  goal-node?
  generate-successors
  If successful search then
    retrieve-explanation
    make-explanation
    make-result 'explained
  else
    make-result 'unmatched
```

- **uniform-cost-search:**
  Implements uniform-cost-search of AND-OR graph:
  1. If start node solved, then search successful.
  2. If list of open nodes is empty, then search has failed.
  3. Remove best node (n) from list of open nodes.
  4. If n is a goal node then mark it solved, and propagate success upwards.
  5. If n is not a goal node, generate its successors and add them to the list of open nodes in such a way that the list remains ordered. If n has no successors, then mark it failed and propagate failure upwards.
  6. Add n to the list of closed nodes.
  7. Go to step 1.

- **generate-successors:**
  Generates the successors of a node for each outgoing relation from the node n.
  1. If there exists a condition on the relation, make sure it is met.
  2. Evaluate the strength of each child node by using the following formula: (child-strength = parent-strength * strength of link connecting them). Modify this strength by using the applicable heuristics (if any).
  3. If child-strength > threshold
     (a) Create AND or OR link in the search graph (depending on the relation)
     (b) Check to see if child node is already on open or closed lists (to prevent infinite loops).
     (c) If necessary generate backsiblings. For example, in (A and B) cause C, B is a backsibling of A, and thus B must be proven before C can be put on open.

- **evaluate-strength:**
  This is the function where heuristics are applied to promote or inhibit certain types of explanations.
• **propagate-success:**
  Given that a child of node \( n \) has been marked solved, mark \( n \) solved if: (1) \( n \) is an or node, or (2) \( n \) is an and node and all its children have been solved.

• **propagate-failure:**
  Given that a child of node \( n \) has been marked failed, mark \( n \) failed if: (1) \( n \) is an and node, or (2) \( n \) is an or node and all its children have failed.

• **retrieve-explanation:**
  Retrieves solution path and stores it in an explanation structure. Beginning with the start node, this function traverses the search graph following the best-child-links and marked graph nodes, storing the explanation in an explanation structure as it goes.

### 9.2.17 reassess-reminders

This function is called when Protos has been reminded of a category which the teacher has rejected, or within which Protos could not find an exemplar matching the new case. The purpose of this function is to reassess the reminders that led to this category, lowering or even removing the reminders. The algorithm is described below.

1. Collect all the reminders to the category that were evoked by features of the new case.
2. Attempt to regenerate an explanation for each of these reminders. If no explanation can be found, remove the reminding, else replace the reminding’s strength with the explanation’s strength.
3. If there are moderate or strong reminders from this same feature to categories which are not known to be subordinates or superordinates of the target category, then decrease the strength of the reminding to the target category.
4. If neither step 2 nor step 3 results in the weakening of the reminding, then decrease the reminding’s strength slightly. If the reminding becomes very weak, remove it.
5. If a feature reminds Protos of at least 2 or 3 categories that have a common superordinate, the reminding is moved to the superordinate and given the strength of the strongest reminding to a subordinate.
6. If a general category with at least 3 subordinate categories shares a reminding with only one of the subordinates, the reminding to the general category is removed.

### 9.2.18 check-term-name

This widely-used utility function takes the name of a term, e.g., *dyspnea* or *fever mild*, and returns the associated instance of the term structure. If the term name is unknown, it will ask the user if the spelling is correct, just to be sure, then it creates the appropriate term structure.
9.2.19 getname

This widely-used utility function takes any named object of the category network (a term, predicate, verb, or quantifier) and returns its print-name. The name that is returned depends on the setting of the global *abbreviation-mode*. If T, it returns the abbreviation for the object if an abbreviation exists; otherwise it returns the full name. The function never returns any synonyms defined for an object.
Chapter 10

Source Files

The CL-Protos distribution tape contains 17 source files comprising the CL-Protos source code. These files are described below in the order in which they are loaded.

10.1 CL-Protos Source Files

- **global.lisp**: As noted in Chapter 2, this file contains global constants and variables used throughout CL-Protos.

- **defs.lisp**: As noted in Chapter 3, this file defines the main structures of the Protos category network, plus their corresponding print functions. Also, other auxiliary structures are defined here, such as the structures for menus, transformations, and matches.

- **verb.lisp**: This file defines the verbs and quantifiers of the Protos explanation language. Each verb and quantifier is represented as an instance of the verb or quantifier structure respectively. Each instance is saved as the value of a distinct global parameter.

- **kbio.lisp**: This file contains all the functions for saving a knowledge base to a file and for loading a knowledge-base from a file. The main top-level functions are `save-kb` and `load-kb`.

- **io.lisp**: This file contains functions for supporting the user interface of CL-Protos. The main functions are `prompt`, `menu-select`, and `escape`.

- **example.lisp**: This file contains the function `load-example` which creates an example category network containing one category (`chairs`) and two exemplars (`chair1`, `chair2`). This example is shown in a diagram in [Bar88, pages 15-16].

- **print.lisp**: This file contains functions for producing various displays of information to the user. Many of these functions display selected portions of the category network. In addition, this file contains some utility functions such as `check-term-name`.

62
10.2. "SYSTEM" FILES

parse.lisp  This file contains: (1) the functions for parsing explanations entered by the user, and (2) the word lists that allow the parser to recognize words and phrases of the Protos explanation language. The top-level function of the parser is get-explanation.

kbpm.lisp  This file contains the knowledge-based pattern matching functions and its main structures (graphnode and graphlink). The top-level function is kbpm.

heuristics.lisp  This file contains heuristics used by kbpm to prefer certain types of explanations and reject or truncate certain other types of explanations.

merging.lisp  This small file contains the functions for testing to see if a case can be merged with an exemplar. There is another set of functions designed to generalize exemplar features when a new case is merged into the exemplar, but all of this code is commented out because Ray decided against generalizing exemplar features.

reminders.lisp  This file contains all the functions associated with processing reminders and censors. The main top-level functions included in this file are: get-raw-reminders, combine-reminders, and re-assess-reminders.

explanation.lisp  This file contains various utility functions for entering and processing explanations (however, the explanation parser itself is in the file parse.lisp). The major functions in this file are: install-relation, enter-explanation, invert-explanation, and compute-explanation-strength.

discuss.lisp  This file contains the functions that Protos uses to discuss a proposed exemplar-to-new-case match with the teacher. The top-level function is discuss-match.

classify.lisp  This file contains most of the functions for Protos' classification algorithm. The top-level function is classify, with subordinates such as build-hypotheses, test-hypotheses, and compare-new-case.

transform.lisp  This small file contains functions for transforming input features before they are used in classification, such as for a quantitative-to-qualitative transformation.

protos.lisp  This file contains the top-level code of CL-Protos including the most prominent menus — Top-Level, Display, Control, Case-Entry, and Unfocused-Instruction. The top-level function is protos, the main function for running CL-Protos.

10.2 “System” Files

The CL-Protos distribution tape contains 3 files that define the Protos system for a Symbolics Genera environment — protos.system, protos.translations, and protos-sys.lisp.
CHAPTER 10. SOURCE FILES

When properly installed in a Symbolics system, these files permit the use of the Genera commands Compile System, Load System, and Edit System. It will be necessary to edit the file protos.translations to set the host name and directory name.

If you're using CL-Protos on any other type of system, use the following file which provides a portable "system" facility.

load-protos.lisp  This file is not part of the CL-Protos program, but rather is a utility program for loading/compiling all the files of CL-Protos. As shown in the CL-Protos Users Guide, when this file is loaded it presents a menu allowing the user to either (1) compile the source files, (2) load the source files for interpretive execution, or (3) load the binary files for execution.
Chapter 11

Miscellaneous

11.1 Extending the Explanation Language

You may wish to add new relations and/or quantifiers to the CL-Protos explanation language. If so, there are two files you must modify: parse.lisp (the explanation parser) and verb.lisp (definitions for relation and quantifier structures). For the sake of example, let’s say that you want to add the new relation creates to the explanation language (plus its inverse is created by). Here is what you would have to do:

1. In file verb.lisp, make instances of the verb structure for creates and is created by as follows:

   (defparameter *verb-creates*
     (make-verb :name "creates" :abbrev "cr" :strength 0.8 :to-proof t :from-proof nil :i-quantifier *quant-sometimes*)

   (defparameter *verb-isCreatedBy*
     (make-verb :name "is created by" :abbrev "icb" :strength 0.8 :to-proof nil :from-proof t :i-quantifier *quant-sometimes* :inverse *verb-creates*)

   (setf (verb-inverse *verb-creates*) *verb-isCreatedBy*)

The value assigned to the to-proof and from-proof slots depends on whether the verb is active, passive, bidirectional, or non-directional. See the comments in the file verb.lisp for an explanation.
2. In the file parse.lisp, two a-lists must be edited. The first, an a-list named verb-list, must include an association between a "standard symbol" for the relation and its corresponding verb instance:

   (creates . *verb-creates*)
   (isCreatedBy . *verb-isCreatedBy*)

3. The second a-list to be edited in parse.lisp is named rel-list, and it contains all the valid ways of specifying a given relation. For creates we must add:

   (creates . creates)
   (cr . creates)

Similar additions must be made to recognize "is created by", "icb", and perhaps "created by".

4. One other list in parse.lisp should also be changed. Named rel-print-list, the contents of this list are displayed only when the user asks for an overview of the relations. This list contains the most common names for each relation (verb) and its inverse.

11.2 Dark Corners

There are a few spots in CL-Protos that haven't been well tested yet, so it's quite possible that you might discover bugs here.

- The facility for transforming features worked at one time but has not been tested as other parts of CL-Protos were changing underneath it. Most of this code is in transform.lisp.

- The code for handling featural exemplars and, more generally, hierarchical concepts, has been tested and seems to work, but it really needs to be more thoroughly tested before it can be trusted. This code is clearly visible in function classify.

- The function reassess-reminders seems to work, but shouldn't be completely trusted until it has been tested on all the different conditions under which it can readjust a reminding.

- The use of conjunctions in the explanation language allows for potentially complex explanations having "backward siblings". For example in the explanation (A and C) causes D, B causes C, if we start from A then we must prove that C is true before concluding D. C is a sibling of A that must be proved in the backward direction, i.e., it is a backward sibling. D is placed on a special list (*limbo*) until C is solved. The code to do this works for the test cases but has never been thoroughly exercised, mainly because backward siblings are rare. The relevant code is in kbpmlisp in the functions add-child-node and generate-backsiblings.
Bibliography


Addendum to "Guide to CL-Protos"

Daniel L. Dvorak

Artificial Intelligence Laboratory
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

November 8, 1988

This note describes a detail of the CL-Protos explanation language that was not explained in chapter 3 of the Guide to CL-Protos. Specifically, there is a distinction between active and passive relations, and this affects the meaning of conjunctions. Consider this explanation:

\[(\text{heat and humidity}) \text{ causes } (\text{muggy-weather and mildew})\]

CL-Protos interprets this to mean that if heat is true and humidity is true then that supports belief in either muggy-weather or mildew, or both. The word "causes" is an active relation, which means that all of the antecedents (heat and humidity) must be true for the relation to hold, and if they are true, then at least one of the consequents (muggy-weather or mildew) will be believed, but it's not necessary to believe all of the consequents in order to accept the explanation. In short, for an active relation, an "and" on the left-hand side is interpreted as a logical-and, but an "and" on the right-hand side is interpreted as a logical-or.

Now let's look at the inverse explanation:

\[(\text{muggy-weather and mildew}) \text{ is caused by } (\text{heat and humidity})\]

The phrase "is caused by" is a passive relation and is the passive counterpart to "causes". For a passive relation only one of the antecedents needs to be true, but all of the consequents must be true for the relation to hold. In other words, heat and humidity must both be true to support belief in either muggy-weather or mildew (or both). Thus, for a passive relation, an "and" on the left-hand side is interpreted as a logical-or and an "and" on the right-hand side is interpreted as a logical-and.

Table 1 shows every active relation and its passive counterpart. Note that the special relation "spurious" is treated as a passive relation where the non-existent consequent is
<table>
<thead>
<tr>
<th>Active</th>
<th>Passive</th>
<th>Bidirectional</th>
<th>Non-directional</th>
</tr>
</thead>
<tbody>
<tr>
<td>implies</td>
<td>is implied by</td>
<td>equivalent</td>
<td>has typical generalization</td>
</tr>
<tr>
<td>causes</td>
<td>is caused by</td>
<td>if and only if</td>
<td>has typical specialization</td>
</tr>
<tr>
<td>enables</td>
<td>is enabled by</td>
<td>definition implies</td>
<td></td>
</tr>
<tr>
<td>has function</td>
<td>is function of</td>
<td>cooccurs with</td>
<td></td>
</tr>
<tr>
<td>has part</td>
<td>is part of</td>
<td>mutually exclusive</td>
<td></td>
</tr>
<tr>
<td>suggests</td>
<td>is inferred from</td>
<td></td>
<td></td>
</tr>
<tr>
<td>is required by</td>
<td>requires</td>
<td></td>
<td></td>
</tr>
<tr>
<td>exhibits</td>
<td>is consistent with</td>
<td></td>
<td></td>
</tr>
<tr>
<td>acts on</td>
<td>is acted on by</td>
<td></td>
<td></td>
</tr>
<tr>
<td>affects</td>
<td>is affected by</td>
<td>spurious</td>
<td></td>
</tr>
</tbody>
</table>

| Must prove all antecedents and at least one consequent. | Must prove all consequents and at least one antecedent. | Must prove all antecedents and all consequents. | Only one antecedent and one consequent allowed. |

Table 1: Relations classified by voice.

assumed to be true. Thus, a spurious relation holds true if any of its antecedents is true. Although the entries for “requires” and “is required by” may appear to be reversed, they are in fact correct. Notice that the sense of cause-and-effect in:

(muggy-weather is caused by (heat and humidity))

is the same as in:

(muggy-weather requires (heat and humidity))

Hence, “requires” is treated as a passive relation for purposes of logic even though it has the grammatical form of an active verb.

In addition to active and passive relations there are also bidirectional and non-directional relations. Bidirectional relations, such as “equivalent”, interpret “and” on both sides of the relation as a logical-and. Bidirectional relations are their own inverses. Non-directional relations allow only a single antecedent and a single consequent. The only two non-directional relations are “has typical generalization” and “has typical specialization”.

2