

Building Expert Systems in Prolog

by

Dennis Merritt



John Stannard on the first free ascent of Foops - 1967

Published by:

Amzi! inc.
5861 Greentree Road
Lebanon, OH 45036 U.S.A.

phone +1-513-425-8050
fax +1-513-425-8025
e-mail info@amzi.com
web www.amzi.com

Book Edition Copyright ©1989 by Springer-Verlag.
On-line Edition Copyright ©2000 by Amzi! inc. All Rights Reserved.

This document ("Work") is protected by copyright laws and international copyright treaties, as well as other intellectual property laws and treaties. You may use and distribute copies of this Work provided each copy of the Work is a true and complete copy, including all copyright and trademark notices, and each copy is accompanied by a copy of this notice. You may not distribute copies of this Work for profit either on a standalone basis or included as part of your own product or work without written permission from Amzi! You may not charge any fees for copies of this work including media or download fees. You may not include this Work as part of your own works. You may not rename, edit or create any derivative works from this Work. Contact Amzi! for additional licensing arrangements.

Amzi! is a registered trademark and Logic Server, Active Prolog Tutor, Adventure in Prolog and the flying squirrel logo are trademarks of Amzi! inc.

Last Updated: August 2000

PDF version March 2001 edited, designed and compiled by Daniel L. Dudley (daniel.dudley@chello.no)

Preface

When I compare the books on expert systems in my library with the production expert systems I know of, I note that there are few good books on building expert systems in Prolog. Of course, the set of actual production systems is a little small for a valid statistical sample, at least at the time and place of this writing – here in Germany, and in the first days of 1989. But there are at least some systems I have seen running in real life commercial and industrial environments, and not only at trade shows.

I can observe the most impressive one in my immediate neighborhood. It is installed in the Telephone Shop of the German Federal PTT near the Munich National Theater, and helps configure telephone systems and small PBXs for mostly private customers. It has a neat, graphical interface, and constructs and prices an individual telephone installation interactively before the very eyes of the customer.

The hidden features of the system are even more impressive. It is part of an expert system network with a distributed knowledge base that will grow to about 150 installations in every Telephone Shop throughout Germany. Each of them can be updated individually overnight via Teletex to present special offers or to adapt the selection process to the hardware supplies currently available at the local warehouses.

Another of these industrial systems supervises and controls in "soft" real time the excavators currently used in Tokyo for subway construction. It was developed on a Unix workstation and downloaded to a single board computer using a real time operating system. The production computer runs exactly the same Prolog implementation that was used for programming, too.

And there are two or three other systems that are perhaps not as showy, but do useful work for real applications, such as oil drilling in the North Sea, or estimating the risks of life insurance for one of the largest insurance companies in the world. What all these systems have in common is their implementation language: Prolog, and they run on "real life" computers like Unix workstations or minis like VAXs. Certainly this is one reason for the preference of Prolog in commercial applications.

But there is one other, probably even more important advantage: Prolog is a programmer's and software engineer's dream. It is compact, highly readable, and arguably the "most structured" language of them all. Not only has it done away with virtually all control flow statements, but even explicit variable assignment too!

These virtues are certainly reason enough to base not only systems but textbooks on this language. Dennis Merritt has done this in an admirable manner. He explains the basic principles, as well as the specialized knowledge representation and processing techniques that are indispensable for the implementation of industrial software such as those mentioned above. This is important because the foremost reason for the relative neglect of Prolog in expert system literature is probably the prejudice that "it can be used only for backward chaining rules." Nothing is farther from the truth. Its relational data base model and its underlying unification mechanism adapt easily and naturally to virtually any programming paradigm one cares to use. Merritt shows how this works using a copious variety of examples. His book will certainly be of particular value for the professional developer of industrial knowledge-based applications, as well as for the student or programmer interested in learning about or building expert systems. I am, therefore, happy to have served as his editor.

Peter H. Schnupp
Munich, January 1989

Acknowledgements

A number of people have helped make this book possible. They include Dave Litwack and Bill Linn of Cullinet who provided the opportunity and encouragement to explore these ideas. Further thanks goes to Park Gerald and the Boston Computer Society, sounding boards for many of the programs in the book. Without the excellent Prolog products from Cogent (now Amzi!), AAIS, Arity, and Logic Programming Associates none of the code would have been developed. A special thanks goes to Peter Gable and Paul Weiss of Arity for their early help and Allan Littleford, provider of both Cogent Prolog and feedback on the book. Jim Humphreys of Suffolk University gave the most careful reading of the book, and advice based on years of experience. As have many other Mac converts, I feel compelled to mention my Macintosh SE, Microsoft Word and Cricket Draw for creating an enjoyable environment for writing books. And finally without both the technical and emotional support of Mary Kroening the book would not have been started or finished.

Table of Contents

Preface	iii
Acknowledgements	iv
1 Introduction	1
1.1 Expert Systems	1
1.2 Expert System Features	3
<i>Goal-Driven Reasoning</i>	3
<i>Uncertainty</i>	4
<i>Data Driven Reasoning</i>	4
<i>Data Representation</i>	5
<i>User Interface</i>	6
<i>Explanations</i>	7
1.3 Sample Applications	7
1.4 Prolog	8
1.5 Assumptions	8
2 Using Prolog's Inference Engine	9
2.1 The Bird Identification System	9
<i>Rule formats</i>	9
<i>Rules about birds</i>	10
<i>Rules for hierarchical relationships</i>	10
<i>Rules for other relationships</i>	11
2.2 User Interface	13
<i>Attribute Value pairs</i>	13
<i>Asking the user</i>	13
<i>Remembering the answer</i>	14
<i>Multi-valued answers</i>	14
<i>Menus for the user</i>	15
<i>Other enhancements</i>	16
2.3 A Simple Shell	16
<i>Command loop</i>	17
<i>A tool for non-programmers</i>	19
2.4 Summary	19
Exercises	19
3 Backward Chaining with Uncertainty	21
3.1 Certainty Factors	21
<i>An Example</i>	21
<i>Rule Uncertainty</i>	22
<i>User Uncertainty</i>	22
<i>Combining Certainties</i>	23
<i>Properties of Certainty Factors</i>	23
3.2 MYCINs Certainty Factors	24
<i>Determining Premise CF</i>	24

<i>Combining Premise CF and Conclusion CF</i>	24
<i>Premise Threshold CF</i>	25
<i>Combining CFs</i>	25
3.3 Rule Format.....	26
3.4 The Inference Engine	27
<i>Working Storage</i>	27
<i>Find a Value for an Attribute</i>	27
<i>Attribute Value Already Known</i>	28
<i>Ask User for Attribute Value</i>	28
<i>Deduce Attribute Value from Rules</i>	28
<i>Negation</i>	30
3.5 Making the Shell.....	30
<i>Starting the Inference</i>	31
3.6 English-like Rules.....	32
Exercises.....	33
4 Explanation	35
<i>Value of Explanations to the User</i>	35
<i>Value of Explanations to the Developer</i>	35
<i>Types of Explanation</i>	36
4.1 Explanation in Clam	36
<i>Tracing</i>	38
<i>How Explanations</i>	39
<i>Why Questions</i>	41
4.2 Native Prolog Systems	43
Exercises.....	46
5 Forward Chaining	47
5.1 Production Systems	47
5.2 Using Oops.....	48
5.3 Implementation.....	52
5.4 Explanations for Oops	56
5.5 Enhancements	56
5.6 Rule Selection	57
<i>Generating the conflict set</i>	57
<i>Time stamps</i>	58
5.7 LEX.....	58
<i>Changes in the Rules</i>	59
<i>Implementing LEX</i>	59
5.8 MEA.....	61
Exercises.....	62
6 Frames.....	65
6.1 The Code.....	66
6.2 Data Structure	66

6.3 The Manipulation Predicates	68
6.4 Using Frames	74
6.5 Summary	75
Exercises	75
7 Integration	77
7.1 Foops (Frames and Oops)	77
<i>Instances</i>	77
<i>Rules for frinsts</i>	79
<i>Adding Prolog to Foops</i>	80
7.2 Room Configuration	81
<i>Furniture frames</i>	82
<i>Frame Demons</i>	83
<i>Initial Data</i>	84
<i>Input Data</i>	85
<i>The Rules</i>	86
<i>Output Data</i>	89
7.3 A Sample Run	90
7.4 Summary	91
Exercises	91
8 Performance	93
8.1 Backward Chaining Indexes	93
8.2 Rete Match Algorithm	94
<i>Network Nodes</i>	95
<i>Network Propagation</i>	96
<i>Example of Network Propagation</i>	97
<i>Performance Improvements</i>	99
8.3 The Rete Graph Data Structures	100
8.4 Propagating Tokens	101
8.5 The Rule Compiler	103
8.6 Integration with Foops	108
8.7 Design Tradeoffs	109
Exercises	109
9 User Interface	111
9.1 Object Oriented Window Interface	111
9.2 Developer's Interface to Windows	111
9.3 High-Level Window Implementation	114
<i>Message Passing</i>	115
<i>Inheritance</i>	115
9.4 Low-Level Window Implementation	117
Exercises	120

10 Two Hybrids	121
10.1 CVGEN.....	121
10.2 The Knowledge Base	122
<i>Rule for parameters.....</i>	122
<i>Rules for derived information.....</i>	123
<i>Questions for the user</i>	124
<i>Default rules.....</i>	124
<i>Rules for edits.....</i>	125
<i>Static information.....</i>	125
10.3 Inference Engine	126
10.4 Explanations.....	127
10.5 Environment	128
10.6 AIJMP	129
10.7 Summary	130
Exercises.....	130
11 Prototyping.....	131
11.1 The Problem.....	131
11.2 The Sales Advisor Knowledge Base	131
<i>Qualifying.....</i>	132
<i>Objectives - Benefits - Features</i>	132
<i>Situation Analysis</i>	133
<i>Competitive Analysis</i>	133
<i>Miscellaneous Advice.....</i>	134
<i>User Queries.....</i>	134
11.3 The Inference Engine	135
11.4 User Interface.....	136
11.5 Summary	138
Exercises.....	138
12 Rubik's Cube.....	139
12.1 The Problem.....	139
12.2 The Cube.....	140
12.3 Rotation	142
12.4 High Level Rules	142
12.5 Improving the State	143
12.6 The Search.....	144
12.7 More Heuristics	145
12.8 User Interface.....	145
12.9 On the Limits of Machines.....	146
Exercises.....	146

Appendices - Full Source Code	147
A Native	149
Birds Knowledgebase (birds.nkb)	149
Native Shell (native.pro)	153
B Clam	157
Car Knowledgebase (car.ckb)	157
Birds Knowledgebase (birds.ckb)	158
Clam Shell (clam.pro)	163
Build Rules (bldrules.pro)	176
C Oops	179
Room Knowledgebase (room.okb)	179
Animal Knowledgebase (animal.okb)	184
Oops Interpreter (oops.pro)	187
D Foops	193
Room Knowledgebase (room.fkb)	193
Foops (foops.pro)	200
E Rete-Foops	211
Room Knowledgebase (room.rkb)	211
Rete Compiler (retepred.pro)	218
Rete Runtime (retefoop.pro)	225
F Windows	239
Windows Demonstration (windemo.pro)	239
Windows (windows.pro)	243
G Rubik	273
Cube Solver (rubik.pro)	273
Cube Display (rubdisp.pro)	286
Cube Entry (rubedit.pro)	289
Move History (rubhist.pro)	291
Moves and Rotations (rubmov.pro)	293
Rubik Help (rubhelp.pro)	296
Rubik Data (rubdata.pro)	297

1 Introduction

Over the past several years there have been many implementations of expert systems using various tools and various hardware platforms, from powerful LISP machine workstations to smaller personal computers.

The technology has left the confines of the academic world and has spread through many commercial institutions. People wanting to explore the technology and experiment with it have a bewildering selection of tools from which to choose. There continues to be a debate as to whether or not it is best to write expert systems using a high-level shell, an AI language such as LISP or Prolog, or a conventional language such as C.

This book is designed to teach you how to build expert systems from the inside out. It presents the various features used in expert systems, shows how to implement them in Prolog, and how to use them to solve problems.

The code presented in this book is a foundation from which many types of expert systems can be built. It can be modified and tuned for particular applications. It can be used for rapid prototyping. It can be used as an educational laboratory for experimenting with expert system concepts.

1.1 Expert Systems

Expert systems are computer applications which embody some non-algorithmic expertise for solving certain types of problems. For example, expert systems are used in diagnostic applications servicing both people and machinery. They also play chess, make financial planning decisions, configure computers, monitor real time systems, underwrite insurance policies, and perform many other services which previously required human expertise.

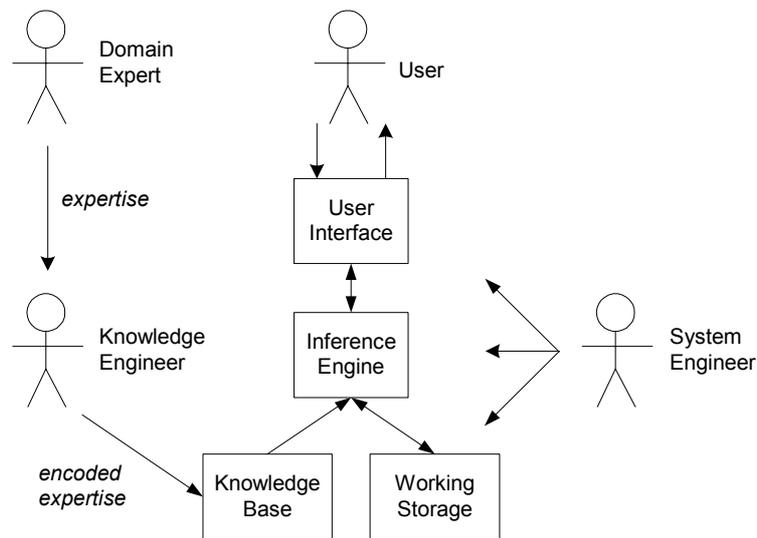


Figure 1.1 Expert system components and human interfaces

Expert systems have a number of major system components and interface with individuals in various roles. These are illustrated in figure 1.1. The major components are:

- Knowledge base – a declarative representation of the expertise, often in IF THEN rules;
- Working storage – the data that is specific to a problem being solved;
- Inference engine – the code at the core of the system, which derives recommendations from the knowledge base and problem-specific data in working storage;
- User interface – the code that controls the dialog between the user and the system.

To understand expert system design, it is also necessary to understand the major roles of individuals who interact with the system. These are:

- Domain expert – the individual or individuals who currently are experts solving the problems the system is intended to solve;
- Knowledge engineer – the individual who encodes the expert's knowledge in a declarative form that can be used by the expert system;
- User – the individual who will be consulting with the system to get advice that would have been provided by the domain expert.

Many expert systems are built with products called expert system shells. The shell is a piece of software which contains the user interface, a format for declarative knowledge in the knowledge base, and an inference engine. The knowledge engineer uses the shell to build a system for a particular problem domain.

Expert systems are also built with shells that are custom developed for particular applications. In this case there is another key individual:

- System engineer – the individual who builds the user interface, designs the declarative format of the knowledge base, and implements the inference engine.

Depending on the size of the project, the knowledge engineer and the system engineer might be the same person. For a custom built system, the design of the format of the knowledge base and the coding of the domain knowledge are closely related. The format has a significant effect on the coding of the knowledge.

One of the major bottlenecks in building expert systems is the knowledge engineering process. The coding of the expertise into the declarative rule format can be a difficult and tedious task. One major advantage of a customized shell is that the format of the knowledge base can be designed to facilitate the knowledge engineering process.

The objective of this design process is to reduce the semantic gap. Semantic gap refers to the difference between the natural representation of some knowledge and the programmatic representation of that knowledge. For example, compare the semantic gap between a mathematical formula and its representation in both assembler and FORTRAN. FORTRAN code (for formulas) has a smaller semantic gap and is therefore easier to work with.

Since the major bottleneck in expert system development is the building of the knowledge base, it stands to reason that the semantic gap between the expert's representation of the knowledge and the representation in the knowledge base should be minimized. With a customized system, the system engineer can implement a knowledge base whose structures are as close as possible to those used by the domain expert.

This book concentrates primarily on the techniques used by the system engineer and knowledge engineer to design customized systems. It explains the various types of inference engines and knowledge bases that can be designed, and how to build and use

them. It tells how they can be mixed together for some problems, and customized to meet the needs of a given application.

1.2 Expert System Features

There are a number of features which are commonly used in expert systems. Some shells provide most of these features, and others just a few. Customized shells provide the features which are best suited for the particular problem. The major features covered in this book are:

- Goal driven reasoning or backward chaining – an inference technique which uses IF THEN rules to repetitively break a goal into smaller sub-goals, which are easier to prove;
- Coping with uncertainty – the ability of the system to reason with rules and data that are not precisely known;
- Data driven reasoning or forward chaining – an inference technique that uses IF THEN rules to deduce a problem solution from initial data;
- Data representation – the way in which the problem specific data in the system is stored and accessed;
- User interface – that portion of the code that creates an easy-to-use system;
- Explanations – the ability of the system to explain the reasoning process that it used to reach a recommendation.

Goal-Driven Reasoning

Goal-driven reasoning, or backward chaining, is an efficient way to solve problems that can be modelled as "structured selection" problems. That is, the aim of the system is to pick the best choice from many enumerated possibilities. For example, an identification problem falls in this category. Diagnostic systems also fit this model, since the aim of the system is to pick the correct diagnosis.

The knowledge is structured in rules, which describe how each of the possibilities might be selected. The rule breaks the problem into sub-problems. For example, the following top level rules are in a system which identifies birds.

```
IF
  family is albatross and
  color is white
THEN
  bird is laysan albatross.
IF
  family is albatross and
  color is dark
THEN
  bird is black footed albatross.
```

The system would try all of the rules which gave information satisfying the goal of identifying the bird. Each would trigger sub-goals. In the case of these two rules, the sub-goals of determining the family and the color would be pursued. The following rule is one that satisfies the family sub-goal:

```
IF
  order is tubenose and
  size large and
  wings long narrow
```

```
THEN
    family is albatross.
```

The sub-goals of determining color, size, and wings would be satisfied by asking the user. By having the lowest level sub-goal satisfied or denied by the user, the system effectively carries on a dialog with the user. The user sees the system asking questions and responding to answers as it attempts to find the rule which correctly identifies the bird.

Uncertainty

Often in structured selection problems the final answer is not known with complete certainty. The expert's rules might be vague, and the user might be unsure of answers to questions. This can be easily seen in medical diagnostic systems where the expert is not able to be definite about the relationship between symptoms and diseases. In fact, the doctor might offer multiple possible diagnoses.

For expert systems to work in the real world they must also be able to deal with uncertainty. One of the simplest schemes is to associate a numeric value with each piece of information in the system. The numeric value represents the certainty with which the information is known. There are numerous ways in which these numbers can be defined, and how they are combined during the inference process.

Data Driven Reasoning

For many problems it is not possible to enumerate all of the possible answers beforehand and have the system select the correct one. For example, system configuration problems fall in this category. These systems might put components in a computer, design circuit boards, or lay out office space. Since the inputs vary and can be combined in an almost infinite number of ways, the goal driven approach will not work.

The data driven approach, or forward chaining, uses rules similar to those used for backward chaining. However, the inference process is different. The system keeps track of the current state of problem solution and looks for rules, which will move that state closer to a final solution.

A system to layout living room furniture would begin with a problem state consisting of a number of unplaced pieces of furniture. Various rules would be responsible for placing the furniture in the room, thus changing the problem state. When all of the furniture was placed, the system would be finished, and the output would be the final state. Here is a rule from such a system which places the television opposite the couch.

```
IF
    unplaced tv and
    couch on wall(X) and
    wall(Y) opposite wall(X)
THEN
    place tv on wall(Y).
```

This rule would take a problem state with an unplaced television and transform it to a state that had the television placed on the opposite wall from the couch. Since the television is now placed, this rule will not fire again. Other rules for other furniture will fire until the furniture arrangement task is finished.

Note that for a data driven system, the system must be initially populated with data, in contrast to the goal driven system which gathers data as it needs it. Figure 1.2 illustrates the difference between forward and backward chaining systems for two simplified rules. The forward chaining system starts with the data of **a=1** and **b=2** and uses the rules to

derive **d=4**. The backward chaining system starts with the goal of finding a value for **d** and uses the two rules to reduce that to the problem of finding values for **a** and **b**.

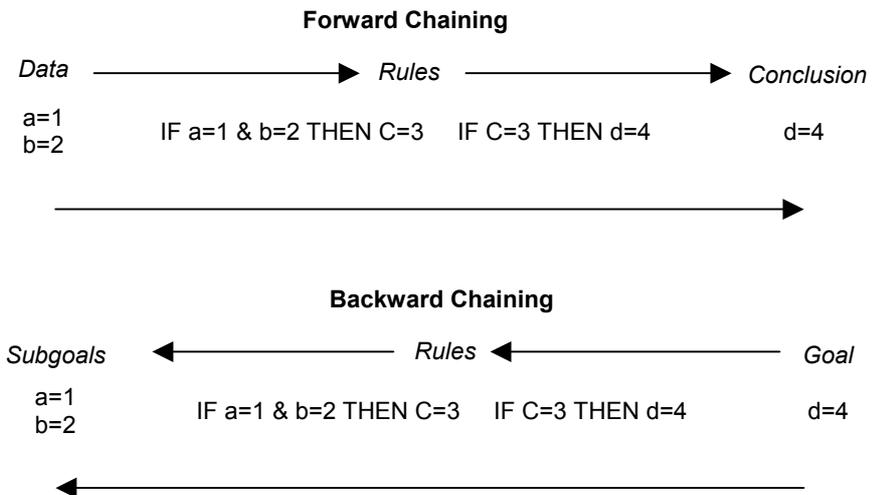


Figure 1.2 Difference between forward and backward chaining

Data Representation

For all rule based systems, the rules refer to data. The data representation can be simple or complex, depending on the problem. The four levels described in this section are illustrated in figure 1.3.

Attribute-Value Pairs

color – white

Object Attribute-Value Triples

arm_chair – width – 3
straight_chair – width – 2

Records

chairs

<i>object</i>	<i>width</i>	<i>color</i>	<i>type</i>
chair #1	3	orange	easy
chair #2	2	brown	straight

Frames

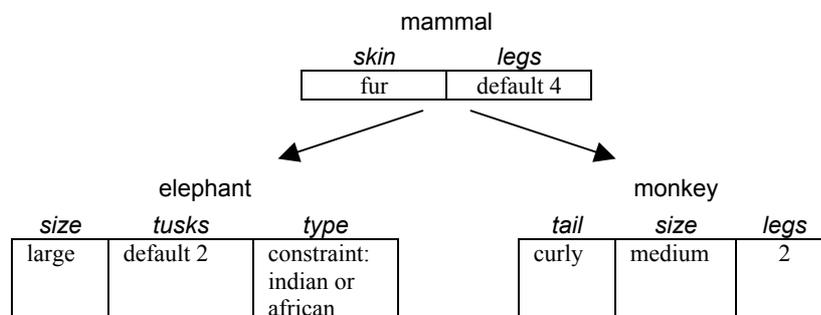


Figure 1.3 Four levels of data representation

The most fundamental scheme uses attribute-value pairs as seen in the rules for identifying birds. Examples are color-white, and size-large.

When a system is reasoning about multiple objects, it is necessary to include the object as well as the attribute-value. For example, the furniture placement system might be dealing with multiple chairs with different attributes, such as size. The data representation in this case must include the object.

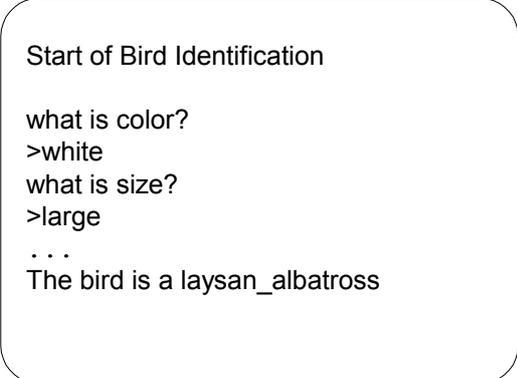
Once there are objects in the system, they each might have multiple attributes. This leads to a record-based structure where a single data item in working storage contains an object name and all of its associated attribute-value pairs.

Frames are a more complex way of storing objects and their attribute-values. Frames add *intelligence* to the data representation, and allow objects to inherit values from other objects. Furthermore, each of the attributes can have associated with it procedures (called demons) which are executed when the attribute is asked for, or updated.

In a furniture placement system each piece of furniture can inherit default values for length. When the piece is placed, demons are activated which automatically adjust the available space where the item was placed.

User Interface

The acceptability of an expert system depends to a great extent on the quality of the user interface. The easiest to implement interfaces communicate with the user through a scrolling dialog as illustrated in figure 1.4. The user can enter commands, and respond to questions. The system responds to commands, and asks questions during the inferencing process.



```
Start of Bird Identification
what is color?
>white
what is size?
>large
...
The bird is a laysan_albatross
```

Figure 1.4 Scrolling dialog user interface

More advanced interfaces make heavy use of pop-up menus, windows, mice, and similar techniques as shown in figure 1.5. If the machine supports it, graphics can also be a powerful tool for communicating with the user. This is especially true for the development interface which is used by the knowledge engineer in building the system.

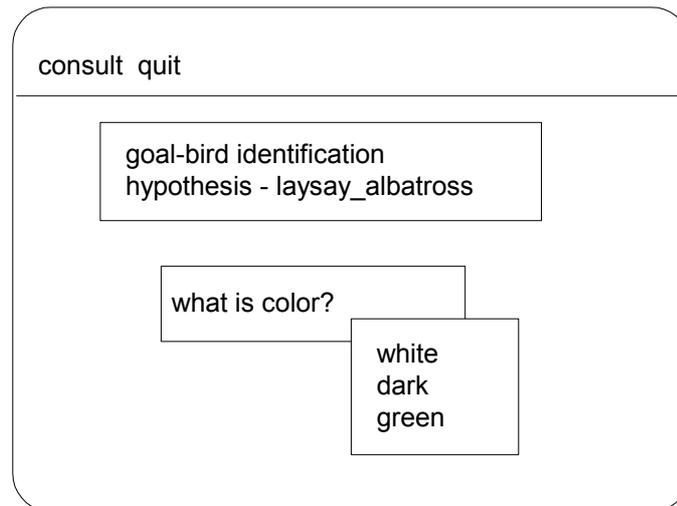


Figure 1.5 Window and menu user interface

Explanations

One of the more interesting features of expert systems is their ability to explain themselves. Given that the system knows which rules were used during the inference process, it is possible for the system to provide those rules to the user as a means for explaining the results.

This type of explanation can be very dramatic for some systems such as the bird identification system. It could report that it knew the bird was a black footed albatross because it knew it was dark colored and an albatross. It could similarly justify how it knew it was an albatross.

At other times, however, the explanations are relatively useless to the user. This is because the rules of an expert system typically represent empirical knowledge, and not a deep understanding of the problem domain. For example a car diagnostic system has rules which relate symptoms to problems, but no rules which describe why those symptoms are related to those problems.

Explanations are always of extreme value to the knowledge engineer. They are the program traces for knowledge bases. By looking at explanations the knowledge engineer can see how the system is behaving, and how the rules and data are interacting. This is an invaluable diagnostic tool during development.

1.3 Sample Applications

In chapters 2 through 9, some simple expert systems are used as examples to illustrate the features and how they apply to different problems. These include a bird identification system, a car diagnostic system, and a system which places furniture in a living room.

Chapters 10 and 11 focus on some actual systems used in commercial environments. These were based on the principles in the book, and use some of the code from the book.

The final chapter describes a specialized expert system which solves Rubik's cube and does not use any of the formalized techniques presented earlier in the book. It illustrates how to customize a system for a highly specialized problem domain.

1.4 Prolog

The details of building expert systems are illustrated in this book through the use of Prolog code. There is a small semantic gap between Prolog code and the logical specification of a program. This means the description of a section of code, and the code are relatively similar. Because of the small semantic gap, the code examples are shorter and more concise than they might be with another language.

The expressiveness of Prolog is due to three major features of the language: rule-based programming, built-in pattern matching, and backtracking execution. The rule-based programming allows the program code to be written in a form which is more declarative than procedural. This is made possible by the built-in pattern matching and backtracking which automatically provide for the flow of control in the program. Together these features make it possible to elegantly implement many types of expert systems.

There are also arguments in favor of using conventional languages, such as C, for building expert system shells. Usually these arguments center around issues of portability, performance, and developer experience. As newer versions of commercial Prologs have increased sophistication, portability, and performance, the advantages of C over Prolog decrease. However, there will always be a need for expert system tools in other languages. (One mainframe expert system shell is written entirely in COBOL.)

For those seeking to build systems in other languages, this book is still of value. Since the Prolog code is close to the logical specification of a program, it can be used as the basis for implementation in another language.

1.5 Assumptions

This book is written with the assumption that the reader understands Prolog programming. If not, *Programming in Prolog* by Clocksin and Mellish from Springer-Verlag is the classic Prolog text. *APT - The Active Prolog Tutor* by the author and published by Solution Systems in South Weymouth, Massachusetts is an interactive PC based tutorial that includes a practice Prolog interpreter.

An in depth understanding of expert systems is not required, but the reader will probably find it useful to explore other texts. In particular since this book focuses on system engineering, readings in knowledge engineering would provide complementary information. Some good books in this area are: *Building Expert Systems* by Hayes-Roth, Waterman, and Lenat; *Rule-Based Expert Systems* by Buchanan and Shortliffe; and *Programming Expert Systems in OPS5* by Brownston, Kant, Farrell, and Martin.

2 Using Prolog's Inference Engine

Prolog has a built-in backward chaining inference engine that can be used to partially implement some expert systems. Prolog rules are used for the knowledge representation, and the Prolog inference engine is used to derive conclusions. Other portions of the system, such as the user interface, must be coded using Prolog as a programming language.

The Prolog inference engine does simple backward chaining. Each rule has a goal and a number of sub-goals. The Prolog inference engine either proves or disproves each goal. There is no uncertainty associated with the results.

This rule structure and inference strategy is adequate for many expert system applications. Only the dialog with the user needs to be improved to create a simple expert system. These features are used in this chapter to build a sample application called "Birds", which identifies birds.

In the later portion of this chapter the Birds system is split into two modules. One contains the knowledge for bird identification, and the other becomes "Native" – the first expert system shell developed in the book. Native can then be used to implement other similar expert systems.

2.1 The Bird Identification System

A system which identifies birds will be used to illustrate a native Prolog expert system. The expertise in the system is a small subset of that contained in *Birds of North America* by Robbins, Bruum, Zim, and Singer. The rules of the system were designed to illustrate how to represent various types of knowledge, rather than to provide accurate identification.

Rule formats

The rules for expert systems are usually written in the form:

```
IF
    first_premise, and
    second_premise, and
    ...
THEN
    conclusion
```

The IF side of the rule is referred to as the left hand side (LHS), and the THEN side is referred to as the right hand side (RHS). This is semantically the same as a Prolog rule:

```
conclusion :-
    first_premise,
    second_premise,
    ...
```

Note that this is a bit confusing since the syntax of Prolog is really THEN IF, and the normal RHS and LHS appear on opposite sides.

Rules about birds

The most fundamental rules in the system identify the various species of birds. We can begin to build the system immediately by writing some rules. Using the normal IF THEN format, a rule for identifying a particular albatross is:

```
IF
  family is albatross and
  color is white
THEN
  bird is laysan_albatross
```

In Prolog the same rule is:

```
bird(laysan_albatross) :-
  family(albatross),
  color(white).
```

The following rules distinguish between two types of albatross and swan. They are clauses of the predicate **bird/1**:

```
bird(laysan_albatross) :-
  family(albatross),
  color(white).
bird(black_footed_albatross) :-
  family(albatross),
  color(dark).
bird(whistling_swan) :-
  family(swan),
  voice(muffled_musical_whistle).
bird(trumpeter_swan) :-
  family(swan),
  voice(loud_trumpeting).
```

In order for these rules to succeed in distinguishing the two birds, we would have to store facts about a particular bird that needed identification in the program. For example if we added the following facts to the program:

```
family(albatross).
color(dark).
```

then the following query could be used to identify the bird:

```
?- bird(X).
X = black_footed_albatross
```

Note that at this very early stage there is a complete working Prolog program, which functions as an expert system to distinguish between these four birds. The user interface is the Prolog interpreter's interface, and the input data is stored directly in the program.

Rules for hierarchical relationships

The next step in building the system would be to represent the natural hierarchy of a bird classification system. These would include rules for identifying the family and the order of a bird. Continuing with the albatross and swan lines, the predicates for **order** and **family** are:

```
order(tubenose) :-
  nostrils(external_tubular),
```

```

        live(at_sea),
        bill(hooked).
order(waterfowl) :-
    feet(webbed),
    bill(flat).
family(albatross) :-
    order(tubenose),
    size(large),
    wings(long_narrow).
family(swan) :-
    order(waterfowl),
    neck(long),
    color(white),
    flight(ponderous).

```

Now the expert system will identify an albatross from more fundamental observations about the bird. In the first version, the predicate for **family** was implemented as a simple fact. Now **family** is implemented as a rule. The *facts* in the system can now reflect more primitive data:

```

nostrils(external_tubular).

live(at_sea).

bill(hooked).

size(large).

wings(long_narrow).

color(dark).

```

The same query still identifies the bird:

```

?- bird(X).
X = black_footed_albatross

```

So far the rules for birds just reflect the attributes of various birds, and the hierarchical classification system. This type of organization could also be handled in more conventional languages as well as in Prolog or some other rule-based language. Expert systems begin to give advantages over other approaches when there is no clear hierarchy, and the organization of the information is more chaotic.

Rules for other relationships

The Canada goose can be used to add some complexity to the system. Since it spends its summers in Canada, and its winters in the United States, its identification includes where it was seen and in what season. Two different rules would be needed to cover these two situations:

```

bird(canada_goose) :-
    family(goose),
    season(winter),
    country(united_states),
    head(black),
    cheek(white).
bird(canada_goose) :-
    family(goose),
    season(summer),
    country(canada),
    head(black),
    cheek(white).

```

These goals can refer to other predicates in a different hierarchy:

```
country(united_states):- region(mid_west).
country(united_states):- region(south_west).
country(united_states):- region(north_west).
country(united_states):- region(mid_atlantic).
country(canada):- province(ontario).
country(canada):- province(quebec).

region(new_england):-
    state(X),
    member(X, [massachusetts, vermont, ...]).
region(south_east):-
    state(X),
    member(X, [florida, mississippi, ...]).
```

There are other birds that require multiple rules for the different characteristics of the male and female. For example the male mallard has a green head, and the female is mottled brown.

```
bird(mallard):-
    family(duck),
    voice(quack),
    head(green).
bird(mallard):-
    family(duck),
    voice(quack),
    color(mottled_brown).
```

Figure 2.1 shows some of the relationships between the rules to identify birds.

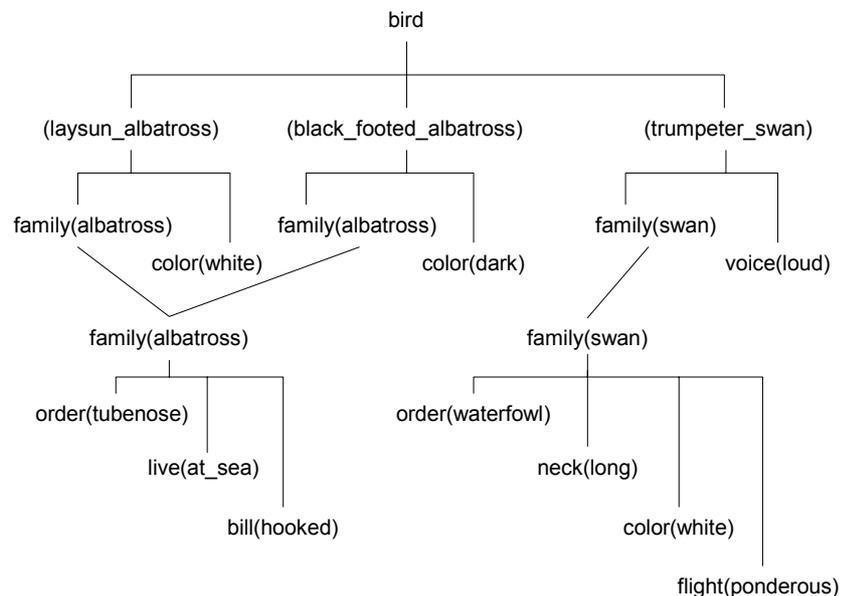


Figure 2.1 Relationships between some of the rules in the Bird identification system

Basically, any kind of identification situation from a bird book can be easily expressed in Prolog rules. These rules form the knowledge base of an expert system. The only drawback to the program is the user interface, which requires the data to be entered into the system as facts.

2.2 User Interface

The system can be dramatically improved by providing a user interface which prompts for information when it is needed, rather than forcing the user to enter it beforehand. The predicate **ask** will provide this functionality.

Attribute Value pairs

Before looking at **ask**, it is necessary to understand the structure of the data which will be asked about. All of the data has been of the form: "attribute-value". For example, a bird is a mallard if it has the following values for these selected bird attributes:

Attribute	Value
family	duck
voice	quack
head	green

This is one of the simplest forms of representing data in an expert system, but is sufficient for many applications. More complex representations can have "object-attribute-value" triples, where the attribute-values are tied to various objects in the system. Still more complex information can be associated with an object and this will be covered in the chapter on frames. For now the simple attribute-value data model will suffice.

This data structure has been represented in Prolog by predicates which use the predicate name to represent the attribute, and a single argument to represent the value. The rules refer to attribute-value pairs as conditions to be tested in the normal Prolog fashion. For example, the rule for mallard had the condition **head(green)** in the rule.

Of course since we are using Prolog, the full richness of Prolog's data structures could be used, as in fact list membership was used in the rules for **region**. The final chapter discusses a system which makes full use of Prolog throughout the system. However, the basic attribute-value concept goes a long way for many expert systems, and using it consistently makes the implementation of features such as the user interface easier.

Asking the user

The **ask** predicate will have to determine from the user whether or not a given attribute-value pair is true. The program needs to be modified to specify which attributes are askable. This is easily done by making rules for those attributes that call **ask**.

```
eats(X):- ask(eats, X).  
feet(X):- ask(feet, X).  
wings(X):- ask(wings, X).  
neck(X):- ask(neck, X).  
color(X):- ask(color, X).
```

Now if the system has the goal of finding **color(white)** it will call **ask**, rather than look in the program. If **ask(color, white)** succeeds, **color(white)** succeeds.

The simplest version of **ask** prompts the user with the requested attribute and value and seeks confirmation or denial of the proposed information. The code is:

```
ask(Attr, Val):-
    write(Attr:Val),
    write('? '),
    read(yes).
```

The **read** will succeed if the user answers "yes", and fail if the user types anything else. Now the program can be run without having the data built into the program. The same query to **bird** starts the program, but now the user is responsible for determining whether some of the attribute-values are true. The following dialog shows how the system runs:

```
?- bird(X).
nostrils : external_tubular? yes.
live : at_sea? yes.
bill : hooked? yes.
size : large? yes.
wings : long_narrow? yes.
color : white? yes.
X = laysan_albatross
```

There is a problem with this approach. If the user answered "no" to the last question, then the rule for **bird(laysan_albatross)** would have failed and backtracking would have caused the next rule for **bird(black_footed_albatross)** to be tried. The first subgoal of the new rule causes Prolog to try to prove **family(albatross)** again, and ask the same questions it already asked. It would be better if the system remembered the answers to questions and did not ask again.

Remembering the answer

A new predicate, **known/3** is used to remember the user's answers to questions. It is not specified directly in the program, but rather is dynamically **asserted** whenever **ask** gets new information from the user.

Every time **ask** is called it first checks to see if the answer is already **known** to be yes or no. If it is not already **known**, then **ask** will **assert** it after it gets a response from the user. The three arguments to **known** are: yes/no, attribute, and value. The new version of **ask** looks like:

```
ask(A, V):-
    known(yes, A, V), % succeed if true
    !. % stop looking
ask(A, V):-
    known(_, A, V), % fail if false
    !,
    fail.
ask(A, V):-
    write(A:V), % ask user
    write('? : '),
    read(Y), % get the answer
    asserta(known(Y, A, V)), % remember it
    Y == yes. % succeed or fail
```

The cuts in the first two rules prevent **ask** from backtracking after it has already determined the answer.

Multi-valued answers

There is another level of subtlety in the approach to **known**. The **ask** predicate now assumes that each particular attribute value pair is either true or false. This means that the user could respond with a "yes" to both color:white and color:black. In effect, we are

letting the attributes be multi-valued. This might make sense for some attributes such as **voice** but not others such as **bill**, which only take a single value.

The best way to handle this is to add an additional predicate to the program, which specifies the attributes that are multi-valued:

```
multivalued(voice).
multivalued(feed).
```

A new clause is now added to **ask** to cover the case where the attribute is not multi-valued (and therefore single-valued) and already has a different value from the one asked for. In this case **ask** should fail. For example, if the user has already answered yes to **size - large** then **ask** should automatically fail a request for **size - small** without asking the user. The new clause goes before the clause which actually asks the user:

```
ask(A, V):-
    not multivalued(A),
    known(yes, A, V2),
    V \== V2,
    !,
    fail.
```

Menus for the user

The user interface can further be improved by adding a menu capability that gives the user a list of possible values for an attribute. It can further enforce that the user enter a value on the menu.

This can be implemented with a new predicate, **menuask**. It is similar to **ask**, but has an additional argument which contains a list of possible values for the attribute. It would be used in the program in an analogous fashion to **ask**:

```
size(X):-
    menuask(size, X, [large, plump, medium, small]).

flight(X):-
    menuask(flight, X, [ponderous, agile, flap_glide]).
```

The **menuask** predicate can be implemented using either a sophisticated windowing interface, or by simply listing the menu choices on the screen for the user. When the user returns a value it can be verified, and the user reprompted if it is not a legal value.

A simple implementation would have initial clauses as in **ask**, and have a slightly different clause for actually asking the user. That last clause of **menuask** might look like:

```
menuask(A, V, MenuList) :-
    write('What is the value for'), write(A), write('?'), nl,
    write(MenuList), nl,
    read(X),
    check_val(X, A, V, MenuList),
    asserta( known(yes, A, X) ),
    X == V.

check_val(X, A, V, MenuList) :-
    member(X, MenuList),
    !.
check_val(X, A, V, MenuList) :-
    write(X), write(' is not a legal value, try again. '), nl,
    menuask(A, V, MenuList).
```

The **check_val** predicate validates the user's input. In this case the test ensures the user entered a value on the list. If not, it retries the **menuask** predicate.

Other enhancements

Other enhancements can also be made to allow for more detailed prompts to the user, and other types of input validation. These can be included as other arguments to **ask**, or embodied in other versions of the **ask** predicate. Chapter 10 gives other examples along these lines.

2.3 A Simple Shell

The bird identification program has two distinct parts: the knowledge base, which contains the specific information about bird identification; and the predicates that control the user interface.

By separating the two parts, a shell can be created, which can be used with any other knowledge base. For example, a new expert system could be written that identifies fish. It could be used with the same user interface code developed for the bird identification system.

The minimal change needed to break the two parts into two modules is a high level predicate that starts the identification process. Since in general it is not known what is being identified, the shell will seek to solve a generic predicate called **top_goal**. Each knowledge base will have to have a **top_goal**, which calls the goal to be satisfied. For example:

```
top_goal(X) :- bird(X).
```

This is now the first predicate in the knowledge base about birds.

The shell has a predicate called **solve**, which does some housekeeping and then solves for the **top_goal**. It looks like:

```
solve :-
    abolish(known, 3),
    define(known, 3),
    top_goal(X),
    write('The answer is '), write(X), nl.
solve :-
    write('No answer found. '), nl.
```

The built-in **abolish** predicate is used to remove any previous **knowns** from the system when a new consultation is started. This allows the user to call **solve** multiple times in a single session.

The **abolish** and **define** predicates are built-in predicates that respectively remove previous **knowns** for a new consultation, and ensure that **known** is defined to the system so no error condition is raised the first time it is referenced. Different dialects of Prolog might require different built-in predicate calls.

In summary, the predicates of the bird identification system have been divided into two modules. The predicates in the shell, called *Native*, are:

- **solve** – starts the consultation;
- **ask** – poses simple questions to the users and remembers the answers;

- **menuask** – presents the user with a menu of choices;
- supporting predicates for the above three predicates.

The predicates in the knowledge base are:

- **top_goal** – specifies the top goal in the knowledge base;
- rules for identifying or selecting whatever it is the knowledge base was built for (for example **bird**, **order**, **family**, and **region**);
- rules for attributes that must be user supplied (for example **size**, **color**, **eats**, and **wings**);
- **multivalued** – defines which attributes might have multiple values.

To use this shell with a Prolog interpreter, both the shell and the birds knowledge base must be consulted. Then the query for **solve** is started.

```
?- consult(native).
yes
?- consult('birds.kb').
yes
?- solve.
nostrils : external_tubular?
...
```

Command loop

The shell can be further enhanced to have a top level command loop called **go**. To begin with, **go** should recognize three commands:

- **load** – Load a knowledge base.
- **consult** – Consult the knowledge base by satisfying the top goal of the knowledge base.
- **quit** – Exit from the shell.

The **go** predicate will also display a greeting and give the user a prompt for a command. After reading a command, **do** is called to execute the command. This allows the command names to be different from the actual Prolog predicates that execute the command. For example, the common command for starting an inference is **consult**; however, **consult** is the name of a built-in predicate in Prolog. This is the code:

```
go :-
    greeting,
    repeat,
        write('> '),
        read(X),
        do(X),
    X == quit.

greeting :-
    write('This is the Native Prolog shell.'), nl,
    write('Enter load, consult, or quit at the prompt.'), nl.

do(load) :-
    load_kb,
    !.
do(consult) :-
    solve,
    !.
```

```

do(quit).
do(X) :-
    write(X),
    write('is not a legal command. '), nl,
    fail.

```

The **go** predicate uses a repeat fail loop to continue until the user enters the command **quit**. The **do** predicate provides an easy mechanism for linking the user's commands to the predicates that do the work in the program. The only new predicate is **load_kb**, which reconsults a knowledge base. It looks like:

```

load_kb :-
    write('Enter file name: '),
    read(F),
    reconsult(F).

```

Two other commands that could be added at this point are:

- **help** – provide a list of legal commands;
- **list** – list all of the **knowns** derived during the consultation (useful for debugging).

This new version of the shell can either be run from the interpreter as before, or compiled and executed. The load command is used to load the knowledge base for use with the compiled shell. The exact interaction between compiled and interpreted Prolog varies from implementation to implementation. Figure 2.2 shows the architecture of the *Native* shell.

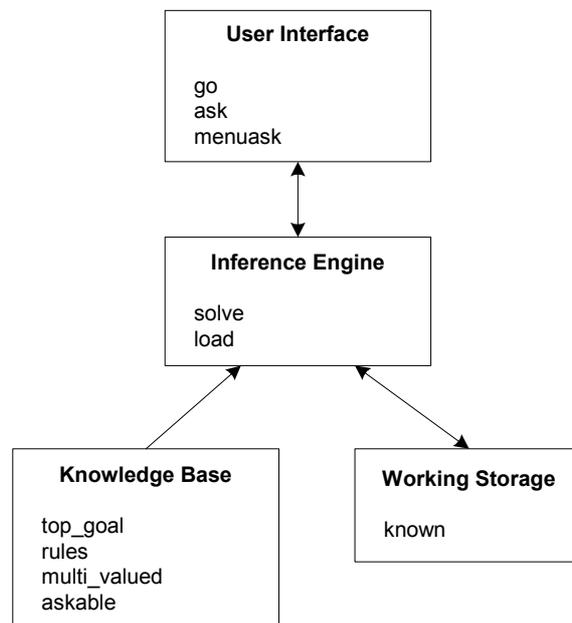


Figure 2.2 Major predicates of Native Prolog shell

Using an interpreter the system would run as follows:

```

?- consult(native).
yes
?- go.
This is the native Prolog shell.
Enter load, consult, or quit at the prompt.
>load.
Enter file name: 'birds.kb'.

```

```
>consult.  
nostrils : external_tubular ? yes.  
...  
The answer is black_footed_albatross  
>quit.  
?-
```

A tool for non-programmers

There are really two levels of Prolog, one which is very easy to work with, and one which is a little more complex.

The first level is Prolog as a purely declarative rule based language. This level of Prolog is easy to learn and use. The rules for bird identification are all formulated with this simple level of understanding of Prolog.

The second level of Prolog requires a deeper understanding of backtracking, unification, and built-in predicates. This level of understanding is needed for the shell.

By breaking the shell apart from the knowledge base, the code has also been divided along these two levels. Even though the knowledge base is in Prolog, it only requires the high level understanding of Prolog. The more difficult parts are hidden in the shell.

This means the knowledge base can be understood with only a little training by an individual who is not a Prolog programmer. In other words, once the shell is hidden from the user, this becomes an expert system tool that can be used with very little training.

2.4 Summary

The example shows that Prolog's native syntax can be used as a declarative language for the knowledge representation of an expert system. The rules lend themselves to solving identification and other types of selection problems that do not require dealing with uncertainty.

The example has also shown that Prolog can be used as a development language for building the user interface of an expert system shell. In this case Prolog is being used as a full programming language.

Exercises

- 2.1 In *Native*, implement commands to provide help and to list the current "known"s.
- 2.2 Have **menuask** print a numbered list of items and let the user just enter the number of the chosen item.
- 2.3 Modify both **ask** and **menuask** to recognize input from the user which is a command, execute the command, and then re-ask the question.
- 2.4 Add a prompt field to **ask** which allows for a longer question for an attribute.
- 2.5 Modify the system to handle attribute-object-value triples as well as attribute-value pairs. For example, rules might have goals such as **color(head, green)**, **color(body, green)**, **length(wings, long)**, and **length(tail, short)**. Now **ask** will prompt with both the object and the attribute as in "head color?". This change will lead to a more natural representation of some of the knowledge in a system as well as reducing the number of attributes.
- 2.6 Use the *Native* shell to build a different expert system. Note any difficulties in implementing the system and features that would have made it easier.

3 Backward Chaining with Uncertainty

As we have seen in the previous chapter, backward chaining systems are good for solving structured selection types of problems. The Birds system was a good example; however, it made the assumption that all information was either absolutely true, or absolutely false. In the real world, there is often uncertainty associated with the rules of thumb an expert uses, as well as the data supplied by the user.

For example, in the Birds system the user might have spotted an albatross at dusk and not been able to clearly tell if it was white or dark colored. An expert system should be able to handle this situation and report that the bird might have been either a laysan or black footed albatross.

The rules too might have uncertainty associated with them. For example, a mottled brown duck might only identify a mallard with 80% certainty.

This chapter will describe an expert system shell, called *Clam*, which supports backward chaining with uncertainty. The use of uncertainty changes the inference process from that provided by pure Prolog, so *Clam* has its own rule format and inference engine.

3.1 Certainty Factors

The most common scheme for dealing with uncertainty is to assign a certainty factor to each piece of information in the system. The inference engine automatically updates and maintains the certainty factors as the inference proceeds.

An Example

Let's first look at an example using *Clam*. The certainty factors (preceded by cf) are integers from -100 (for definitely false) to +100 (for definitely true).

The following is a small knowledge base in *Clam* that is designed to diagnose a car which will not start. It illustrates some of the behavior of one scheme for handling uncertainty.

```
goal problem.

rule 1
if not turn_over and
battery_bad
then problem is battery.

rule 2
if lights_weak
then battery_bad cf 50.

rule 3
if radio_weak
then battery_bad cf 50.

rule 4
if turn_over and
smell_gas
then problem is flooded cf 80.

rule 5
if turn_over and
gas_gauge is empty
then problem is out_of_gas cf 90.
```

```

rule 6
if turn_over and
gas_gauge is low
then problem is out_of_gas cf 30.

ask turn_over
menu (yes no)
prompt 'Does the engine turn over?'.

ask lights_weak
menu (yes no)
prompt 'Are the lights weak?'.

ask radio_weak
menu (yes no)
prompt 'Is the radio weak?'.

ask smell_gas
menu (yes no)
prompt 'Do you smell gas?'.

ask gas_gauge
menu (empty low full)
prompt 'What does the gas gauge say?'.

```

The inference uses backward chaining similar to pure Prolog. The goal states that a value for the attribute **problem** is to be found. Rule 1 will cause the sub-goal of **bad_battery** to be pursued – just as in Prolog.

The rule format also allows for the addition of certainty factors. For example rules 5 and 6 reflect the varying degrees of certainty with which one can conclude that the car is out of gas. The uncertainty arises from the inherent uncertainty in gas gauges. Rules 2 and 3 both provide evidence that the battery is bad, but neither one is conclusive.

Rule Uncertainty

What follows is a sample dialog of a consultation with the Car expert system.

```

consult, restart, load, list, trace, how, exit
:consult
Does the engine turn over?
:yes
Do you smell gas?
:yes
What does the gas gauge say?
empty
low
full
:empty
problem-out_of_gas-cf-90
problem-flooded-cf-80
done with problem

```

Notice that, unlike Prolog, the inference does not stop after having found one possible value for problem. It finds all of the reasonable problems and reports the certainty to which they are known. As can be seen, these certainty factors are not probability values, but simply give some degree of weight to each answer.

User Uncertainty

The following dialog shows how the user's uncertainty might be entered into the system. The differences from the previous dialog are shown in bold.

```

:consult
Does the engine turn over?
: yes
Do you smell gas?
: yes cf 50
What does the gas gauge say?
empty
low
full
: empty
problem-out_of_gas-cf-90
problem-flooded-cf-40
done with problem

```

Notice in this case that the user was only certain to a degree of 50 that there was a gas smell. This results in the system only being half as sure that the **problem** is **flooded**.

Combining Certainties

Finally consider the following consultation, which shows how the system combines evidence for a bad battery. Remember that there were two rules that concluded the battery was weak with a certainty factor of 50.

```

:consult
Does the engine turn over?
: no
Are the lights weak?
: yes
Is the radio weak?
: yes
problem-battery-cf-75
done with problem

```

In this case the system combined the two rules to determine that the battery was weak with certainty factor 75. This propagated straight through rule 1 and became the certainty factor for **problem battery**.

Properties of Certainty Factors

There are various ways in which the certainty factors can be implemented, and how they are propagated through the system, but they all have to deal with the same basic situations:

- rules whose conclusions are uncertain;
- rules whose premises are uncertain;
- user entered data which is uncertain;
- combining uncertain premises with uncertain conclusions;
- updating uncertain working storage data with new, also uncertain information;
- establishing a threshold of uncertainty for when a premise is considered known.

Clam uses the certainty factor scheme that was developed for MYCIN, one of the earliest expert systems used to diagnose bacterial infections. Many commercial expert system shells today use this same scheme.

3.2 MYCINs Certainty Factors

The basic MYCIN certainty factors (CFs) were designed to produce results that seemed intuitively correct to the experts. Others have argued for factors that are based more on probability theory and still others have experimented with more complex schemes designed to better model the real world. The MYCIN factors, however, do a reasonable job of modeling for many applications with uncertain information.

We have seen from the example how certainty information is added to the rules in the **then** clause. We have also seen how the user can specify CFs with input data. These are the only two ways uncertainty gets into the system.

Uncertainty associated with a particular run of the system is kept in working storage. Every time a value for an attribute is determined by a rule or a user interaction, the system saves that attribute value pair and associated CF in working storage.

The CFs in the conclusion of the rule are based on the assumption that the premise is known with a CF of 100. That is, if the conclusion has a CF of 80 and the premise is known to CF 100, then the fact which is stored in working storage has a CF of 80. For example, if working storage contained:

```
turn_over cf 100
smell_gas cf 100
```

then a firing of rule 4:

```
rule 4
if turn_over and
smell_gas
then problem is flooded cf 80
```

would result in the following fact being added to working storage:

```
problem flooded cf 80
```

Determining Premise CF

However, it is unlikely that a premise is perfectly known. The system needs a means for determining the CF of the premise. The algorithm used is a simple one. The CF for the premise is equal to the minimum CF of the individual sub goals in the premise. If working storage contained:

```
turn_over cf 80
smell_gas cf 50
```

then the premise of rule 4 would be known with CF 50, the minimum of the two.

Combining Premise CF and Conclusion CF

When the premise of a rule is uncertain due to uncertain facts, and the conclusion is uncertain due to the specification in the rule, then the following formula is used to compute the adjusted certainty factor of the conclusion:

$$CF = RuleCF * PremiseCF / 100.$$

Given the above working storage and this formula, the result of a firing of rule 4 would be:

```
problem is flooded cf 40
```

The resulting CF has been appropriately reduced by the uncertain premise. The premise had a certainty factor of 50, and the conclusion a certainty factor of 80, thus yielding an adjusted conclusion CF of 40.

Premise Threshold CF

A threshold value for a premise is needed to prevent all of the rules from firing. The number 20 is used as a minimum CF necessary to consider a rule for firing. This means that if working storage had:

```
turn_over cf 80
smell_gas cf 15
```

then rule 4 would not fire due to the low CF associated with the premise.

Combining CFs

Next, consider the case where there is more than one rule that supports a given conclusion. In this case, each of the rules might fire and contribute to the CF of the resulting fact. If a rule fires supporting a conclusion, and that conclusion is already represented in working memory by a fact, then the following formulae are used to compute the new CF associated with the fact. X and Y are the CFs of the existing fact and rule conclusion.

```
CF(X, Y) = X + Y(100 - X)/100.  X, Y both > 0
CF(X, Y) = X + Y/1 - min(|X|, |Y|).  one of X, Y < 0
CF(X, Y) = -CF(-X, -Y).  X, Y both < 0
```

For example, both rules 2 and 3 provide evidence for **battery_bad**:

```
rule 2
if lights_weak
then battery_bad cf 50.

rule 3
if radio_weak
then battery_bad cf 50.
```

Assume the following facts are in working storage:

```
lights_weak cf 100
radio_weak cf 100
```

A firing of rule 2 would then add the following fact:

```
battery_bad cf 50
```

Next, rule 3 would fire – also concluding **battery_bad** cf 50. However, there already is a **battery_bad** fact in working storage, so rule 3 updates the existing fact with the new conclusion using the formulae above. This results in working storage being changed to:

```
battery_bad cf 75
```

This case most clearly shows why a new inference engine was needed for *Clam*. When trying to prove a conclusion for which the CF is less than 100, we want to gather all of the

evidence both for and against that conclusion and combine it. Prolog's inference engine will only look at a single rule at a time, and succeed or fail based on it.

3.3 Rule Format

Since we are writing our own inference engine, we can design our own internal rule format as well. (We will use something easier to read for the user.) It has at least two arguments: one for the IF or left hand side (LHS), which contains the premises, and one for the THEN or right hand side (RHS), which contains the conclusion. It is also useful to keep a third argument for a rule number or name. The overall structure looks like:

```
rule(Name, LHS, RHS).
```

The name will be a simple atom identifying the rule. The **LHS** and **RHS** must hold the rest of the rule. Typically in expert systems, a rule is read: LHS implies RHS. This is backwards from a Prolog rule, which can be thought of as being written RHS :- LHS, or RHS is implied by LHS. That is, the RHS (conclusion) is written on the left of the rule, and the LHS (premises) is written on the right.

Since we will be backward chaining, and each rule will be used to prove or disprove some bit of information, the **RHS** contains one goal pattern, and its associated CF. This is:

```
rhs(Goal, CF)
```

The **LHS** can have many sub-goals, which are used to prove or disprove the **RHS**:

```
lhs(GoalList)
```

where **GoalList** is a list of goals.

The next bit of design has to do with the actual format of the goals themselves. Various levels of sophistication can be added to these goals, but for now we will use the simplest form, which is attribute-value pairs. For example, **gas_gauge** is an attribute, and **low** is a value. Other attributes have simple yes-no values, such as **smell_gas**. An attribute-value pair will look like:

```
av(Attribute, Value)
```

where Attribute and Value are simple atoms. The entire rule structure looks like:

```
rule(Name,
      lhs( [av(A1, V1), av(A2, V2), ...] ),
      rhs( av(Attr, Val), CF) ).
```

Internally, rule 5 looks like:

```
rule(5,
      lhs( [av(turns_over, yes), av(gas_gauge, empty)] ),
      rhs( av(problem, flooded), 80) ).
```

This rule format is certainly not easy to read, but it makes the structure clear for programming the inference engine. There are two ways to generate more readable rules for the user. One is to use operator definitions. The other is to use Prolog's language handling ability to parse our own rule format. The built-in definite clause grammar (DCG) of most Prologs is excellent for this purpose. Later in this chapter we will use DCG to create a clean user interface to the rules. The forward chaining system in a later chapter uses the operator definition approach.

3.4 The Inference Engine

Now that we have a format for rules, we can write our own inference engine to deal with those rules. Let's summarize the desired behavior:

- combine certainty factors as indicated previously;
- maintain working storage information that is updated as new evidence is acquired;
- find all information about a particular attribute when it is asked for, and put that information in working storage.

The major predicates of the inference engine are shown in figure 3.1. They are described in detail in the rest of this section.

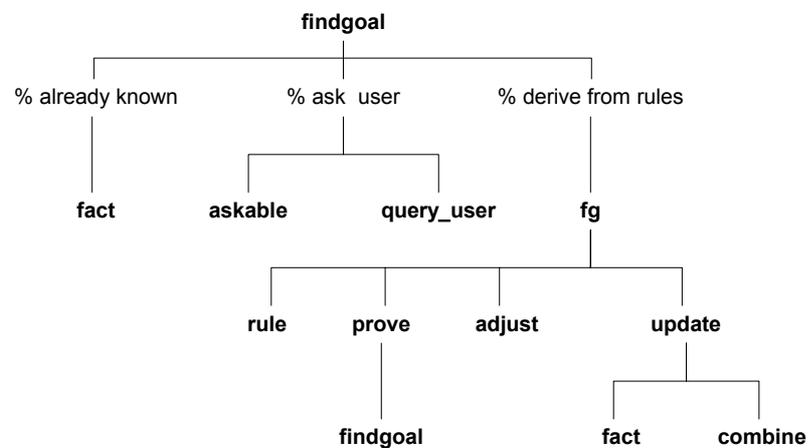


Figure 3.1 Major predicates of Clam inference engine

Working Storage

Let's first decide on the working storage format. It will simply contain the known facts about attribute-value pairs. We will use the Prolog database for them and store them as:

```
fact( av(A, V), CF ).
```

Find a Value for an Attribute

We want to start the inference by asking for the value of a goal. In the case of the Car expert system we want to find the value of the attribute **problem**. The main predicate that does inferencing will be **findgoal/2**. In the Car expert system, it could be called from an interpreter with the following query:

```
?- findgoal( av(problem, X), CF ).
```

The **findgoal/2** predicate has to deal with three distinct cases:

- the attribute-value is already known;
- there are rules to deduce the attribute-value;
- we must ask the user.

The system can be designed to automatically ask the user if there are no rules, or it can be designed to force the knowledge engineer to declare which attribute values will be supplied by the user. The latter approach makes the knowledge base for the expert system more explicit, and also provides the opportunity to add more information controlling the dialog with the user. This might be in the form of clearer prompts, and/or input validation criteria.

We can define a new predicate **askable/2** that tells which attributes should be retrieved from the user, and the prompt to use. For example:

```
askable(live, 'Where does it live?').
```

With this new information we can now write **findgoal**.

Attribute Value Already Known

The first rule covers the case where the information is in working storage. It was asserted so we know all known values of the attribute have been found. Therefore we cut, so no other clauses are tried.

```
findgoal(av(Attr, Val), CF) :-
    fact(av(Attr, Val), CF),
    !.
```

Ask User for Attribute Value

The next rule covers the case where there is no known information, and the attribute is askable. In this case we simply ask.

```
findgoal(av(Attr, Val), CF) :-
    not fact(av(Attr, _), _),
    askable(Attr, Prompt),
    query_user(Attr, Prompt),
    !,
    findgoal(av(Attr, Val), CF).
```

The **query_user** predicate prompts the user for a value and CF and then asserts it as a **fact**. The recursive call to **findgoal** will now pick up this **fact**.

```
query_user(Attr, Prompt) :-
    write(Prompt),
    read(Val),
    read(CF),
    asserta(fact(av(Attr, Val), CF)).
```

Deduce Attribute Value from Rules

The final rule of **findgoal** covers the interesting case of using other rules. Remember that the inferencing is going to require looking for all rules that provide support for values for the sought attribute, and combining the CFs from them. This is done by calling **fg**, which uses a repeat fail loop to continue to find rules whose RHS conclude a value for the attribute. The process stops when the attribute is known with a CF of 100, or all the rules have been tried.

```
findgoal(Goal, CurCF) :-
    fg(Goal, CurCF).
```

```

fg(Goal, CurCF) :-
    rule(N, lhs(IfList), rhs(Goal, CF)),
    prove(IfList, Tally),
    adjust(CF, Tally, NewCF),
    update(Goal, NewCF, CurCF),
    CurCF == 100,
    !.
fg(Goal, CF) :-
    fact(Goal, CF).

```

The three new predicates called in **fg** are as follows:

- **prove** – prove the LHS premise and find its CF;
- **adjust** – combine the LHS CF with the RHS CF;
- **update** – update existing working storage values with the new conclusion.

This is the guts of the inferencing process for the new inference engine. First it finds a rule whose RHS matches the pattern of the goal. It then feeds the LHS of that rule into **prove**, which succeeds if the LHS can be proved. The **prove** predicate returns the combined CF from the LHS. If **prove** fails, backtracking is initiated causing the next rule, which might match the goal pattern, to be tried.

```

prove(IfList, Tally) :-
    prov(IfList, 100, Tally).

prov([], Tally, Tally).
prov([H|T], CurTal, Tally) :-
    findgoal(H, CF),
    min(CurTal, CF, Tal),
    Tal >= 20,
    prov(T, Tal, Tally).

min(X, Y, X) :-
    X <= Y,
    !.
min(X, Y, Y) :-
    Y <= X.

```

The input argument to **prove** is the list of premises for the rule; the output is the **Tally**, or combined CF from the premises. The **prove** predicate calls **prov** with an extra argument to keep track of **Tally**. At each recursion the **Tally** is reset to the minimum up to that point. Of course, **prov** recursively calls **findgoal** for each of the premises. Notice the check to make sure the **Tally** stays above 20. This is the threshold value for considering an attribute-value pair to be true.

After **prove** succeeds, **adjust** computes the combined CF based on the RHS CF and the **Tally** from the LHS.

```

adjust(CF1, CF2, CF) :-
    X is CF1 * CF2 / 100,
    int_round(X, CF).

int_round(X, I) :-
    X >= 0,
    I is integer(X + 0.5).
int_round(X, I) :-
    X < 0,
    I is integer(X - 0.5).

```

Then **update** combines the new evidence for this attribute-value pair with any existing known evidence. The first argument is the attribute-value pair just deduced, and the second

is its CF. The third argument is the returned value for the CF when it is combined with existing facts for the attribute-value pair.

```
update(Goal, NewCF, CF) :-
    fact(Goal, OldCF),
    combine(NewCF, OldCF, CF),
    retract( fact(Goal, OldCF) ),
    asserta( fact(Goal, CF) ),
    !.
update(Goal, CF, CF) :-
    asserta( fact(Goal, CF) ).

combine(CF1, CF2, CF) :-
    CF1 >= 0,
    CF2 >= 0,
    X is CF1 + CF2*(100 - CF1)/100,
    int_round(X, CF).
combine(CF1, CF2, CF) :-
    CF1 < 0,
    CF2 < 0,
    X is - ( -CF1 -CF2 * (100 + CF1)/100),
    int_round(X, CF).
combine(CF1, CF2, CF) :-
    (CF1 < 0; CF2 < 0),
    (CF1 > 0; CF2 > 0),
    abs_minimum(CF1, CF2, MCF),
    X is 100 * (CF1 + CF2) / (100 - MCF),
    int_round(X, CF).
```

The **abs_minimum** predicate finds the minimum in terms of absolute value. The code can be seen in the appendix.

Negation

One last point is to deal with negation. The premises might also be of the form **not goal**. In this case we call **findgoal** for the **goal**, and complement the CF to find the degree of certainty of its negation. For example, if a fact has a CF of 70, then **not** fact has a certainty of -70.

```
findgoal(not Goal, NCF) :-
    findgoal(Goal, CF),
    NCF is - CF,
    !.
```

This rule should become the first clause for **findgoal**.

3.5 Making the Shell

Now that the inference engine is built, it can become part of a shell. The code to build this first version of the *Clam* shell is the same as that used to build the *Native* shell. It consists of a command loop with the commands **load**, **consult**, and **exit**. Figure 3.2 shows the architecture of *Clam*.

```
super :-
    repeat,
    write('consult, load, exit'), nl,
    write(':'),
    read_line(X),
    doit(X),
    X == exit.
```

```

doit(consult) :-
    top_goals,
    !.
doit(load) :-
    load_rules,
    !.
doit(exit).

```

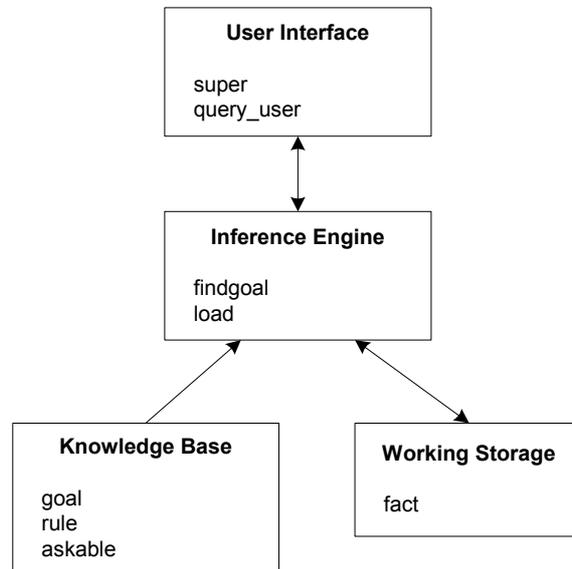


Figure 3.2 Major predicates of the Clam shell

Starting the Inference

The **consult** command does a little more than just call **findgoal**. It calls **top_goals** which uses the **top_goal** facts to start the inference. The system might have more than one **top_goal** to allow sequencing of the consultation. For example a diagnostic system might have two goals, the first diagnoses the problem, and the second recommends a solution.

After **top_goals** satisfies a goal, it prints the values for the goal as seen in the early examples of Car.

```

top_goals :-
    top_goal(Attr),
    top(Attr),
    print_goal(Attr),
    fail.
top_goals.

top(Attr) :-
    findgoal(av(Attr, Val), CF),
    !.
top(_) :-
    true.

print_goal(Attr) :-
    nl,
    fact(av(Attr, X), CF),
    CF >= 20,
    outp(av(Attr, X), CF), nl,
    fail.
print_goal(Attr) :-
    write('done with '), write(Attr), nl, nl.

```

```

outp(av(A, V), CF) :-
    output(A, V, PrintList),
    write(A-'cf'-CF),
    printlist(PrintList),
    !.
outp(av(A, V), CF) :-
    write(A-V-'cf'-CF).

printlist([]).
printlist([H|T]) :-
    write(H),
    printlist(T).

```

3.6 English-like Rules

The **load** command for *Clam* does not simply read in Prolog terms as in *Native*, but instead uses DCG to read in a knowledge base in the format shown earlier in the chapter for the Car system. You might notice that the askable items have the additional syntax to allow menu choices, which was not included in the implementation details above. It is coded similarly to the menu feature in *Native*.

The **load_kb** predicate in the shell gets a file name as in *Native*, and then calls **load_rules** with the file name.

```

load_rules(F) :-
    clear_db,
    see(F),
    lod_ruls,
    write('rules loaded'), nl,
    seen,
    !.
lod_ruls :-
    repeat,
        read_sentence(L),
        process(L),
    L == eof.

process(eof) :- !.
process(L) :-
    trans(R, L, []),
    assertz(R),
    !.
process(L) :-
    write('translate error on:'), nl,
    write(L), nl.

clear_db :-
    abolish(top_goal, 1),
    abolish(askable, 4),
    abolish(rule, 3).

```

This section of code basically calls **read_sentence** to tokenize a sentence (up to a ".") into a list. The token list is then processed by the DCG predicate **trans**, and the resulting Prolog term, **R**, is asserted in the knowledge base. For a good description of DCG, see Clocksin & Mellish chapter 9, *Using Grammar Rules*. The **clear_db** predicate removes all earlier **top_goal**, **askable**, and **rule** predicates so that a new knowledge base can be loaded over an existing one.

The tokenizing predicate, **read_sentence**, varies from Prolog to Prolog based on the implementation. If the implementation has built-in predicates that can read tokens, then **read_sentence** is trivial. If not, it has to read the input character by character and build the

tokens. An example of this type of sentence read predicate can be found in Clocksin & Mellish section 5.3, *Reading English Sentences*.

The top level DCG predicate, **trans**, looks for the three types of statements allowed in the knowledge base:

```
trans(top_goal(X))-->[goal, is, X].
trans(top_goal(X))-->[goal, X].
trans(askable(A, M, P))-->
    [ask, A], menux(M), prompt(A, P).
trans(rule(N, lhs(IF), rhs(THEN, CF)))-->
    id(N),
    if(IF),
    then(THEN, CF).
```

The following predicates recognize the menu and prompt fields in the ask statement.

```
menux(M)--> [menu, '(', menuxlist(M).

menuxlist([Item])--> [Item, ')'].
menuxlist([Item|T])--> [Item], menuxlist(T).

prompt(_, P)--> [prompt, P].
prompt(P, P)--> [].
```

Next are the predicates used to parse the basic rule structure. Note the flexibility that can be added into the system with DCG. Both **and** and **,** can be used as LHS separators. The attribute-value phrases can be expressed in many different ways to allow the most natural expression in the rules.

```
id(N)--> [rule, N].

if(IF)--> [if], iflist(IF).

iflist([IF])--> phrase(IF), [then].
iflist([Hif|Tif])--> phrase(Hif), [and], iflist(Tif).
iflist([Hif|Tif])--> phrase(Hif), [' ', '], iflist(Tif).

then(THEN, CF)--> phrase(THEN), [cf], [CF].
then(THEN, 100)--> phrase(THEN).

phrase(not av(Attr, yes))--> [not, Attr].
phrase(not av(Attr, yes))--> [not, a, Attr].
phrase(not av(Attr, yes))--> [not, an, Attr].
phrase(not av(Attr, Val))--> [not, Attr, is, Val].
phrase(not av(Attr, Val))--> [not, Attr, are, Val].
phrase(av(Attr, Val))--> [Attr, is, Val].
phrase(av(Attr, Val))--> [Attr, are, Val].
phrase(av(Attr, yes))--> [Attr].
```

Using DCG in this fashion, it is easy to implement as friendly a syntax for the knowledge base as desired. The same approach could also be used for the *Native* system, with DCG that translated English-like rules into Prolog syntax.

Now that we have a customized knowledge base and inference engine, it is possible to add other features to the system. The next chapter shows how to add explanations.

Exercises

3.1 Add attribute object value triples to the knowledge representation of *Clam*.

- 3.2 There are other ways of dealing with uncertainty in the literature, which could be used with *Clam*. A simple one would just use a few text modifiers such as weak, very weak, or strong and have rules for combining them. Implement this or some other scheme in *Clam*.
- 3.3 Isolate the predicates that are used for calculating certainty factors, such that it is easy to add additional methods. Implement them so the calling predicates do not need to know the syntax of the certainty factor, since they might be text, numbers, or more complex data structures.
- 3.4 Allow rules to have optional threshold values associated with them, which override the default of 20. This would be an addition to the rule syntax as well as the code.
- 3.5 Have the inference engine automatically generate a prompt to the user when there is no **askable** or **rule** that finds a value for an attribute.
- 3.6 Add menus to the query user facility.
- 3.7 Implement another diagnostic application using *Clam*. Note any difficulties and features which could be added to alleviate those difficulties.

4 Explanation

It is often claimed that an important aspect of expert systems is the ability to explain their behavior. This means the user can ask the system for justification of conclusions or questions at any point in a consultation with an expert system. The system usually responds with the rules that were used for the conclusion, or the rules being considered that led to a question to the user.

Value of Explanations to the User

The importance of this feature is probably overestimated for the user. Typically the user just wants the answer. Furthermore, when the user does want an explanation, the explanation is not always useful. This is due to the nature of the "intelligence" in an expert system.

The rules typically reflect empirical, or "compiled" knowledge. They are codifications of an expert's rules of thumb, not the expert's deeper understanding that led to the rules of thumb. For example, consider the following dialog with an expert system designed to give advice on car problems:

```
Does the car start? no.
Does the engine turn over? yes.
Do you smell gas? yes.
Recommendation - Wait 5 minutes and try again.
why?
I used the rule:
If not start, and
engine_turn_over, and
smell_gas
Then recommend is 'Wait 5 minutes and try again.'
```

The rule gives the correct advice for a flooded car, and knows the questions to ask to determine if the car is flooded, but it does not contain the knowledge of what a flooded car is and why waiting will help. If the user really wanted to understand what was happening, he/she would need a short dissertation on carburetors, how they behave, and their relationship to the gas pedal.

For a system such as this to have useful explanations, it would need to do more than parrot the rules used. One approach is to annotate the rules with deeper explanations. This is illustrated in chapter 10. Another approach being actively researched is to encode the deeper knowledge into the system and use it to drive both the inference and the explanations.

On the other hand, there are some systems in which the expert's knowledge is just empirical knowledge. In this case, the system's explanation is useful to the user. Classification systems such as the bird identification system fall in this category. The Bird system would explain an identification of a laysan albatross with the rule used to identify it. There is no underlying theory as to why a white albatross is a laysan albatross and a dark one is a black footed albatross. That is simply the rule used to classify them.

Value of Explanations to the Developer

While an explanation feature might be of questionable value to the user of the system, it is invaluable to the developer of the system. It serves the same diagnostic purpose as program tracing for conventional programs. When the system is not behaving correctly, the expert can use the explanations to find the rules that are in error. The knowledge

engineer uses the explanations to better tune the knowledge base to have more realistic dialogs with the user.

Types of Explanation

There are four types of explanations commonly used in expert systems. We will implement most of these in both the *Clam* shell and the *Native* shell:

- a rule trace, which reports on the progress of a consultation;
- explanation of **how** the system reached a given conclusion;
- explanation of **why** the system is asking a question;
- explanation of **why not** a given conclusion.

Since we wrote the inference engine for *Clam* it will not be difficult to modify it to include these features. The *Native* system currently uses Prolog's inference engine. In order to add explanation it will be necessary to write our own Prolog inference engine. Fortunately it is not difficult to write Prolog in Prolog.

4.1 Explanation in Clam

First, let's look at some examples of the explanation features of *Clam* using the Car system. Here is how the user turns on tracing for the consultation, and the results. The new trace information is in bold. It shows the sequence of rule firings as they are expected. Notice in particular that it reports correctly on the nesting of rules 2 and 3 within rule 1.

```
consult, restart, load, list, trace, how, exit
:trace on
consult, restart, load, list, trace, how, exit
:consult
:call rule 1
Does the engine turn over?
: no
:call rule 2
Are the lights weak?
: yes
:exit rule 2
:call rule 3
Is the radio weak?
: yes
:exit rule 3
:exit rule 1
:call rule 4
:fail rule 4
:call rule 5
:fail rule 5
:call rule 6
:fail rule 6
problem-battery-cf-75
done with problem
```

Next we can look at the use of **why** explanations. The user would ask why and get the inference chain that led to the question. For example:

```
...
Is the radio weak?
:why
rule 3
If
```

```

radio_weak
Then
battery_bad 50
rule 1
If
not turn_over
battery_bad
Then
problem is battery 100
goal problem
...

```

Notice that the why explanation gives the chain of rules, in reverse order, that led to the question. In this case the **goal problem** led to rule 1 which led to rule 3.

The **how** explanations start with answers. For example, the system has just reported that the problem is the battery. The user wants to know how this result was derived.

```

...
problem-battery-cf-75
done with problem
consult, restart, load, list, trace, how, exit
:how
Goal? problem is battery
problem is battery was derived from rules: 1
rule 1
If
not turn_over
battery_bad
Then
problem is battery 100

```

In this case the rule(s) which directly supported the result are listed. Next the user wants to know how **battery_bad** was derived.

```

consult, restart, load, list, trace, how, exit
:how
Goal? battery_bad
battery_bad was derived from rules: 3 2
rule 3
If
radio_weak
Then
battery_bad 50
rule 2
If
lights_weak
Then
battery_bad 50

```

In this case there were two rules that supported the goal, and the system lists them both.

Figure 4.1 shows the difference between **how** and **why** questions. The **why** questions occur at the bottom of an inference chain, and the **how** questions occur at the top.

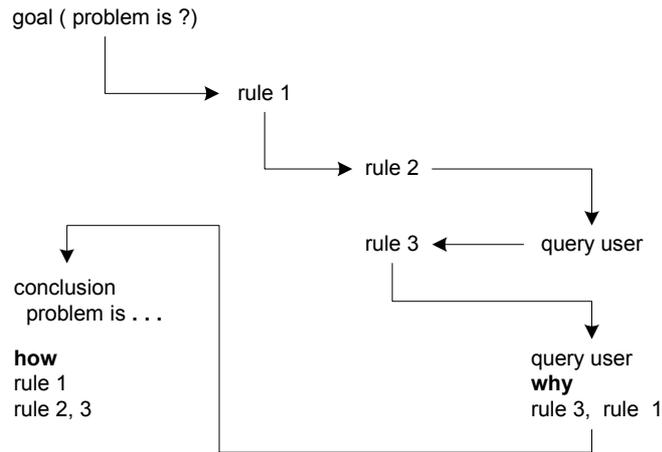


Figure 4.1. Difference between how and why questions

Tracing

The first explanation addition to *Clam* will be the rule tracing facility. It will behave similarly to the Prolog box model traces, and inform the user when a rule is "call"ed, "exit"ed, or "fail"ed. It will use a special predicate **bugdisp** to communicate trace information with the user. It will take as an argument a list of terms to be written on a line.

To make it a user option, **bugdisp** will only write if **ruletrace** is true. The user will have a new high level command to turn tracing on or off which will assert or retract **ruletrace**. We can then use **bugdisp** to add any diagnostics printout we like to the program.

```

bugdisp(L) :-
    ruletrace,
    write_line(L),
    !.
bugdisp(_).

write_line([]) :-
    nl.
write_line([H|T]) :-
    write(H),
    tab(1),
    write_line(T).
  
```

Here is the new command added to the **do** predicate called by the command loop predicate, **go**. It allows the user to turn tracing on or off by issuing the command **trace(on)** or **trace(off)**.

```

do( trace(X) ) :-
    set_trace(X),
    !.

set_trace(off) :-
    ruletrace,
    retract( ruletrace ).
set_trace(on) :-
    not ruletrace,
    asserta( ruletrace ).
set_trace(_).
  
```

Now that we have the tools for displaying trace information, we need to add **bugdisp** calls in the predicate that recursively tries rules, **fg**. It is easy to determine in **fg** when a rule is called and when it has been successful. After the call to **rule** succeeds, the rule has been called. After the call to **prove**, the rule has been successfully fired. The new code for the predicate is added in bold.

```
fg(Goal, CurCF) :-
    rule(N, lhs(IfList), rhs(Goal, CF)),
    bugdisp(['call rule', N]),
    prove(N, IfList, Tally),
    bugdisp(['exit rule', N]),
    adjust(CF, Tally, NewCF),
    update(Goal, NewCF, CurCF, N),
    CurCF == 100,
    !.
fg(Goal, CF) :-
    fact(Goal, CF).
```

All that remains is to capture rules that fail after being called. The place to do this is in a second clause to **prove**, which is called when the first clause fails. The second clause informs the user of the failure, and continues to fail.

```
prove(N, IfList, Tally) :-
    prov(IfList, 100, Tally),
    !.
prove(N, _, _) :-
    bugdisp(['fail rule', N]),
    fail.
```

How Explanations

The next explanation feature to implement is *how*. The how question is asked by the user to see the proof of some conclusion the system has reached. The proof can be generated by either rederiving the result with extra tracing, or by having the original derivation stored in working storage. *Clam* uses the second option and stores derivation information with the **fact** in working storage. Each fact might have been derived from multiple rules, all concluding the same attribute value pair and combining certainty factors. For this reason, a list of rule numbers is stored as the third argument to **fact**. This is not the entire proof tree, but just those rules which conclude the fact directly.

```
fact(AV, CF, RuleList)
```

A **fact** is updated by **update**, so this is where the derivation is captured. A new argument is added to **update** which is the rule number that caused the update. Note that the first clause of **update** adds the new rule number to the list of existing derivation rule numbers for the **fact**. The second clause merely creates a new list with a single element.

```
update(Goal, NewCF, CF, RuleN) :-
    fact(Goal, OldCF, _),
    combine(NewCF, OldCF, CF),
    retract( fact(Goal, OldCF, OldRules) ),
    asserta( fact(Goal, CF, [RuleN | OldRules] ) ),
    !.
update(Goal, CF, CF, RuleN) :-
    asserta( fact(Goal, CF, [RuleN] ) ).
```

The call to update from **fg** is modified to fill in the new argument with a rule number:

```
fg(Goal, CurCF) :-
    rule(N, lhs(IfList), rhs(Goal, CF)),
    ...
```

```

    update(Goal, NewCF, CurCF, N),
    ...

```

Now that the supporting rules for each derived fact are in working storage, we can answer a user's question about how a fact was derived. The simplest thing to do is to have **how** simply write the list of rules used. It is probably of more interest to the user to actually display the rules as well. The predicate **list_rules** does that.

```

how(Goal) :-
    fact(Goal, CF, Rules),
    CF > 20,
    pretty(Goal, PG),
    write_line([PG, was, derived, from, 'rules: '|Rules]),
    nl,
    list_rules(Rules),
    fail.
how(_).

```

The **how** predicate for negated goals is similar and uses the fact that negation is represented by a negative CF.

```

how(not Goal) :-
    fact(Goal, CF, Rules),
    CF < -20,
    pretty(not Goal, PG),
    write_line([PG, was, derived, from, 'rules: '|Rules]),
    nl,
    list_rules(Rules),
    fail.

```

The **pretty** predicate is used to convert **av** structures into a more readable list and visa versa.

```

pretty(av(A, yes), [A]) :- !.
pretty(not av(A, yes), [not, A]) :- !.
pretty(av(A, no), [not, A]) :- !.
pretty(not av(A, V), [not, A, is, V]).
pretty(av(A, V), [A, is, V]).

```

The **list_rules** predicate writes a formatted listing of each rule used in deriving a given fact.

```

list_rules([]).
list_rules([R|X]) :-
    list_rule(R),
    list_rules(X).

list_rule(N) :-
    rule(N, lhs(Iflist), rhs(Goal, CF)),
    write_line(['rule ', N]),
    write_line(['If']),
    write_ifs(Iflist),
    write_line(['Then']),
    pretty(Goal, PG),
    write_line([' ', PG, CF]), nl.

write_ifs([]).
write_ifs([H|T]) :-
    pretty(H, HP),
    tab(5), write_line(HP),
    write_ifs(T).

```

We can use **pretty** in reverse, along with a predicate that reads a list of tokens from a line to provide a nicer interface to the user for how questions. In this way the user doesn't have to specify the internal form of the fact.

```
how :-
    write('Goal? '), read_line(X), nl,
    pretty(Goal, X),
    how(Goal).
```

The **how** command can now be added as part of the top level user interface:

```
do(how) :-
    how,
    !.
```

The full **how** command as coded above just displays for the user the rules directly responsible for a fact. These rules themselves are likely based on other facts, which were derived as well. There are two ways of presenting this information:

- let the user ask further **hows** of the various rules' left hand side goals to delve deeper into the proof tree;
- have **how** automatically display the entire proof tree.

So far we have chosen the first. In order to implement the second choice, a predicate **how_lhs** needs to be written, which will trace the full tree by recursively calling **how** for each of the goals in the **Iflist** of the rule.

```
list_rules([]).
list_rules([R|X]) :-
    list_rule(R),
    how_lhs(R),
    list_rules(X).

how_lhs(N) :-
    rule(N, lhs(Iflist), _),
    !,
    how_ifs(Iflist).

how_ifs([]).
how_ifs([Goal|X]) :-
    how(Goal),
    how_ifs(X).
```

The three choices of user interface for **hows** (just rule numbers, listings of direct rules, list of full proof tree) shows some of the problems with shells and the advantages of a toolbox approach. In a customized expert system, the options that make the most sense for the application can be used. In a generalized system the designer is faced with two unpleasant choices. One is to keep the system easy to use and pick one option for all users. The other is to give the flexibility to the user and provide all three, thus making the product more complex for the user to learn.

Why Questions

The **how** question is asked from the top level of an inference, after the inference has been completed. The **why** question is asked at the bottom of a chain of rules — when there are no more rules and it is time to ask the user. The user wants to know **why** the question is being asked.

In order to be able to answer this type of question, we must keep track of the inference chain that led to the question to the user. One way to do this is to keep an extra argument in the inference predicates that contains the chain of rules above it in the inference. This is done in **findgoal** and **prove**. Each keeps a separate argument, **Hist**, which is the desired list of rules. The list is initially the empty list at the top call to **findgoal**.

```
findgoal(Goal, CurCF, Hist) :-
    fg(Goal, CurCF, Hist).

fg(Goal, CurCF, Hist) :-
    ...
    prove(N, IfList, Tally, Hist),
    ...
```

The **prove** predicate maintains the list by adding the current rule number on the head of the list before a recursive call to **findgoal**. The calls further down the recursion have this new rule number available for answers to **why** questions. Notice that both Prolog's recursive behavior and backtracking assure that the history is correct at any level of call.

```
prove(N, IfList, Tally, Hist) :-
    prov(IfList, 100, Tally, [N|Hist]),
    !.
prove(N, _, _) :-
    bugdisp(['fail rule', N]),
    fail.

prov([], Tally, Tally, Hist).
prov([H|T], CurTal, Tally, Hist) :-
    findgoal(H, CF, Hist),
    min(CurTal, CF, Tal),
    Tal >= 20,
    prov(T, Tal, Tally, Hist).
```

Finally, we need to give the user the ability to ask the *why* question without disturbing the dialog. This means replacing the old **reads** of user input with a new predicate, **get_user**, which gets an answer from the user and processes it as a **why** command if necessary. **Hist** is of course passed down as an argument and is available for **get_user** to process. Also, rather than just displaying rule numbers, we can list the rules for the user as well.

The **process_ans** predicate first looks for command patterns and behaves accordingly. If it is a command, the command is executed and then failure is invoked causing the system to backtrack and re-ask the user for an answer.

Note that now that we are capturing and interpreting the user's response with more intelligence, we can give the user more options. For example, at the question level he/she can turn tracing on or off for the duration of the session, ask a how question, or request help. These are all easily added options for the implementer.

```
get_user(X, Hist) :-
    repeat,
        write(': '),
        read_line(X),
        process_ans(X, Hist).

process_ans([why], Hist) :-
    nl, write_hist(Hist),
    !,
    fail.
process_ans([trace, X], _) :-
    set_trace(X),
    !,
    fail.
process_ans([help], _) :-
```

```

    help,
    !,
    fail.
process_ans(X, _). % just return user's answer

write_hist([]) :-
    nl.
write_hist([goal(X)|T]) :-
    write_line([goal, X]),
    !,
    write_hist(T).
write_hist([N|T]) :-
    list_rule(N),
    !,
    write_hist(T).

```

4.2 Native Prolog Systems

Since we wrote the inference engine for *Clam*, it was easy to modify it to add the code for explanations. However, when we use pure Prolog, we don't have access to the inference engine.

This problem is easily solved. We simply write a Prolog inference engine in Prolog. Then, having written the inference engine, we can modify it to handle explanations.

An inference engine has to have access to the rules. In Prolog, the clauses are themselves just Prolog terms. The built-in predicate **clause** gives us access to the rules. It has two arguments, which unify with the head of a clause and its body. A fact has a body with just the goal true.

Predicates that manipulate Prolog clauses are confusing to read due to the ambiguous use of the comma in Prolog. It can be either:

- an operator used to separate the subgoals in a clause; or
- a syntactic separator of functor arguments.

Prolog clauses are just Prolog terms with functors of ":-" and ",". Just for now, pretend Prolog used an "&" operator to separate goals rather than a "," operator. Then a clause would look like:

```
a :- b & c & d.
```

Without the operator definitions it would look like:

```
:- (a, &(b, &(c, d))).
```

The **clause** built-in predicate picks up the first and second arguments of the ":-" functor. It will find the entire Prolog database on backtracking. If patterns are specified in either argument, then only clauses that unify with the patterns are found. For the above clause:

```
?- clause(Head, Body).
Head = a
Body = b & c & d
```

A recursive predicate working through the goals in **Body** would look like:

```

recurse(FirstGoal & RemainingGoals) :-
    process(FirstGoal),
    recurse(RemainingGoals).
recurse(SingleGoal) :-
    process(SingleGoal).

```

The use of "&" was just to distinguish between the two commas in Prolog. To resolve ambiguous references to commas as in the first line of the above code, parenthesis are used. The first line should really be written:

```

recurse( (FirstGoal, RemainingGoals) ) :-
...

```

 See Clocksin & Mellish Section 2.3, *Operators* for a full discussion of operators.

Given the means to access and manipulate the Prolog database of facts and rules, a simple Prolog interpreter that proves a list of goals (goals separated by the "," operator) would look like:

```

prove(true) :- !.
prove((Goal, Rest)) :-
    clause(Goal, Body),
    prove(Body),
    prove(Rest).
prove(Goal) :-
    clause(Goal, Body),
    prove(Body).

```

Notice that **prove** mimics precisely Prolog's behavior. First it finds a clause whose head matches the first goal. Then it proves the list of goals in the **Body** of the clause. Notice that unification automatically occurs between the **Goal** for the head of the clause and the **Body**. This is because the Prolog clause is just a Prolog term. If it succeeds, it continues with the rest of the goals in the list. If it fails, it backtracks and finds the next clause whose head unifies with the **Goal**.

This interpreter will only handle pure Prolog whose clauses are asserted in the database. It has no provisions for built-in predicates. These could be included by adding a final catchall clause:

```

prove(X) :- call(X).

```

For *Native* we do not intend to have Prolog built-in predicates, but we do intend to call **ask** and **menuask**. For the *Native* shell these are our own built-in predicates.

We will make some basic modifications to our Prolog interpreter to allow it to handle our own built-in predicates and record information for explanations. First, we write an intermediate predicate **prov** that calls **clause**. It can also check for built-in predicates such as **ask** and **menuask** in the system. If the goal is either of these, they are just called with real Prolog.

Next we add an extra argument, just as we did for *Clam*. The extra argument keeps track of the level of nesting of a particular goal. By passing this history along to the **ask** predicates, the **ask** predicates can now respond to **why** questions.

```

prove(true, _) :- !.
prove((Goal, Rest), Hist) :-
    prov(Goal, (Goal, Rest)),
    prove(Rest, Hist).

prov(true, _) :- !.

```

```

prov(menuask(X, Y, Z), Hist) :-
    menuask(X, Y, Z, Hist),
    !.
prov(ask(X, Y), Hist) :-
    ask(X, Y, Hist),
    !.
prov(Goal, Hist) :-
    clause(Goal, List),
    prove(List, [Goal|Hist]).

```

Notice that the history is a list of goals, and not the full rules as saved in *Clam*.

The next step is to modify the top level predicate that looks for birds. First add an empty history list as an argument to the top call of prove:

```

solve :-
    abolish(known, 3),
    define(known, 3),
    prove(top_goal(X), []),
    write('The answer is '), write(X), nl.
solve :-
    write('No answer found'), nl.

```

The processing of **why** questions is the same as in *Clam*.

```

get_user(X, Hist) :-
    repeat,
    read(X),
    process_ans(X, Hist),
    !.

process_ans(why, Hist) :-
    write(Hist),
    !,
    fail.
process_ans(X, _) .

```

The dialog with the user would look like:

```

?- identify.
nostrils : external_tubular? why.
[nostrils(external_tubular), order(tubenose), family(albatross),
bird(laysan_albatross)]
nostrils : external_tubular?

```

We can further use **clause** to answer **how** questions. In *Clam* we chose to save the derivations in the database. For native Prolog it is easier just to rederive the answer.

```

how(Goal) :-
    clause(Goal, List),
    prove(List, []),
    write(List).

```

It is also possible to ask **whynot** questions, which determine why an expected result was not reached. This also uses **clause** to find the clauses that might have proved the goals, and goes through the list of goals looking for the first one that failed. It is reported, and then backtracking causes any other clauses that might have helped to be explained as well.

```

whynot(Goal) :-
    clause(Goal, List),
    write_line([Goal, 'fails because: ']),
    explain(List).

```

```

whynot(_).

explain( (H, T) ) :-
    check(H),
    explain(T).
explain(H) :-
    check(H).

check(H) :-
    prove(H, _),
    write_line([H, succeeds]),
    !.
check(H) :-
    write_line([H, fails]),
    fail.

```

The **whynot** predicate has the same design problems as **how**. Do we automatically recurse through a whole failure tree, or do we let the user ask successive **whynot**'s to delve deeper into the mystery. This version just gives the first level. By adding a recursive call to **whynot** in the second clause of **check**, it would print the whole story.

Exercises

- 4.1 Implement **whynot** for *Clam*.
- 4.2 Have **whynot** give a full failure history.
- 4.3 Make sure the explanation facility can handle attribute object value triples in both *Clam* and *Native*.
- 4.4 Decide whether you like the full rules presented in answer to **why** questions as in *Clam*, or just the goals as in *Native*. Make both systems behave the same way.
- 4.5 Enhance the trace function so it displays the goals currently being sought by the system. Have various levels of trace information that can be controlled by the trace command.
- 4.6 Using **prove**, implement a Prolog trace function.
- 4.7 Add a pretty printing predicate for *Native* to use when displaying Prolog rules.

5 Forward Chaining

This chapter discusses a forward chaining rule based system and its expert system applications. It shows how the forward chaining system works, how to use it, and how to implement it quickly and easily using Prolog.

A large number of expert systems require the use of forward chaining, or data driven inference. The most famous of these is Digital Equipment Corporation's XCON system. It configures computers. It starts with the data about the customer order and works forward toward a configuration based on that data. The XCON system was written in the OPS5 (forward chaining rule based) language.

Data driven expert systems are different from the goal driven, or backward chaining systems seen in the previous chapters.

The goal driven approach is practical when there are a reasonable number of possible final answers, as in the case of a diagnostic or identification system. The system methodically tries to prove or disprove each possible answer, gathering the needed information as it goes.

The data driven approach is practical when combinatorial explosion creates a seemingly infinite number of possible right answers, such as possible configurations of a machine.

5.1 Production Systems

Forward chaining systems are often called “production” systems. Each of the rules is actually a miniature procedure called a production. When the patterns in the left hand side match working storage elements, then the actions on the right hand side are taken. This chapter concentrates on building a production system called *Oops*.

Production systems are composed of three components. These are:

- the rule set;
- a working storage area which contains the current state of the system;
- an inference engine which knows how to apply the rules.

The rules are of the form:

```
left hand side (LHS) ==> right hand side (RHS).
```

The LHS is a collection of conditions which must be matched in working storage for the rule to be executed. The RHS contains the actions to be taken if the LHS conditions are met.

The execution cycle is:

1. Select a rule whose left hand side conditions match the current state as stored in the working storage.
2. Execute the right hand side of that rule, thus somehow changing the current state.
3. Repeat until there are no rules which apply.

Production systems differ in the sophistication of the algorithm used to select a rule (step 1). The first version of *Oops* will use the simplest algorithm of just selecting the first rule whose conditions match working storage.

The knowledge engineer programs in *Oops* by creating rules for the particular application. The syntax of the rules is:

```
rule <rule id>:
  [<N>: <condition>, .....]
  ==>
  [<action>, ....].
```

where:

rule id — a unique identifier for the rule;
N — optional identification for the condition;
condition — a pattern to match against working storage;
action — an action to take.

Each **condition** is a legal Prolog data structure, including variables. Variables are identified, as in Prolog, by an initial upper case letter, or underscore. In general, the **condition** pattern is matched against those stored in working storage. The comparison operators (>, =<, etc.) are also defined for comparing variables that are bound during the pattern matching.

Note that the data representation of *Oops* is richer than that used in *Clam*. In *Clam* there were only attribute–value pairs, or object–attribute–value triples. In *Oops* the data can be represented by any legal Prolog term including variables.

The following RHS actions are supported:

assert(X) — adds the term *X* to working storage;
retract(all) — retracts all of the working storage terms which were matched in the LHS of the rule being executed;
retract(N) — retracts LHS condition number *N* from working storage;
X = <arithmetic expression> — sets *X* to the value of the expression;
X # Y — unifies the structures *X* and *Y*;
write(X) — writes the term *X* to the terminal;
nl — writes a new line at the terminal;
read(X) — reads a term and unifies it to *X*;
prompt(X, Y) — writes *X* and reads *Y*;

5.2 Using Oops

In the Winston & Horn LISP book there is an example of a forward chaining animal identification system. Some of those rules would be expressed in *Oops* like this:

```
rule id6:
  [1: has(X, pointed_teeth),
   2: has(X, claws),
   3: has(X, forward_eyes)]
  ==>
```

```
[retract(all),
 assert(isa(X, carnivore))].
```

This rule would fire if working storage contained the Prolog terms:

```
has(robie, pointed_teeth)
has(robie, claws)
has(robie, forward_eyes)
```

When the rule fired, these three terms would be removed by the **retract(all)** action on the RHS, and would be replaced with the term:

```
isa(robie, carnivore)
```

Since the working storage elements which matched the conditions were removed, this rule would not fire again. Instead, another rule such as this might fire next:

```
rule id10:
[1: isa(X, mammal),
 2: isa(X, carnivore),
 3: has(X, black_stripes)]
==>
[retract(all),
 assert(isa(X, tiger))].
```

Rules about relationships are also easily coded. Again from Winston & Horn's example the rule that says children are the same type of animal as their parents is expressed as follows:

```
rule id16:
[1: isa(Animal, Type),
 2: parent(Animal, Child)]
==>
[retract(2),
 assert(isa(Child, Type))].
```

This would transform working storage terms like:

```
isa(robie, tiger)
parent(robie, suzie)
```

to:

```
isa(robie, tiger)
isa(suzie, tiger)
```

The working storage is initialized with a statement of the form:

```
initial_data([<term>, .....]).
```

Each term is a legal Prolog term, which is asserted in working storage. For example:

```
initial_data([gives(robie, milk),
eats_meat(robie),
has(robie, tawny_color),
has(robie, dark_spots),
parent(robie, suzie)]).
```

It would be better if we could set up the system to ask the user for the initial terms. In a conventional programming language we might set up a loop which repeatedly asked for data until the user typed in "end".

To do the same thing in a production system requires a bit of trickery, which goes against the grain of rule based systems. Theoretically, the rules are independent and don't communicate with each other, but by setting flags in working storage the programmer can force a specific order of rule firings.

Here is how to code the input loop in *Oops*. It violates another tenet of production systems by making use of the known rule selection strategy. In the case of *Oops*, it is known that rule 1 will be tried before rule 2.

```
initial_data([read_facts]).

rule 1:                % This is the end condition of
[1: end,              % the loop. If "end" and
 2: read_facts]       % "read_facts" are both
                    % in working storage,
==>
[retract(all)].      % then remove them both.

rule 2:                % This is the body of the loop.
[1: read_facts]       % If "read_facts" is in WS,
==>                  % then prompt for an attribute
[prompt('Attribute? ', X), % and assert it. If X
 assert(X)].          % is "end", rule 1 will fire next.
```

The animal identification problem is one that can be solved either in a data driven (forward chaining) approach as illustrated here and in Winston & Horn, or in a goal driven (backward chaining) approach. In fact, where the list of possible animals is known the backward chaining approach is probably a more natural one for this problem.

A more suitable problem for a forward chaining system is configuration. The *Oops* sample program in the appendix shows such a system. It lays out furniture in a living room.

The program includes a number of rules for laying out furniture. For example:

- The couch goes on the longer wall of the room, and is not on the same side as the door.
- The television goes opposite the couch.
- If there is a lamp or television on a wall without a plug, buy an extension cord.

Here are those rules in *Oops*. They are more complex due to the need to work with the amount of wall space available.

```
% f1 - the couch goes opposite the door
rule f1:
[1: furniture(couch, LenC), % an unplaced couch
  position(door, DoorWall), % find the wall with the door
  opposite(DoorWall, OW), % the wall opposite the door
  right(DoorWall, RW), % the wall right of the door
 2: wall(OW, LenOW), % available space opposite
  wall(RW, LenRW), % available space to the right
  LenOW >= LenRW, % if opposite bigger than right
  LenC <= LenOW] % length of couch less than
                % wall space
==>
[retract(1), % remove the furniture term
 assert(position(couch, OW)), % assert new position
 retract(2), % remove the old wall, length
 NewSpace = LenOW - LenC, % calculate the space left
 assert(wall(OW, NewSpace))]. % assert new space

% f3 - the tv should be opposite the couch
```

```

rule f3:
[1: furniture(tv, LenTV),
 2: position(couch, CW),
 3: opposite(CW, W),
 4: wall(W, LenW),
   LenW >= LenTV]
==>
[retract(1),
  assert(position(tv, W)),
  retract(4),
  NewSpace = LenW - LenTV,
  assert(wall(W, NewSpace))].

% get extension cords if needed
rule f12:
[1: position(tv, W),
 2: not(position(plug, W))]
==>
[assert(buy(extension_cord, W)),
  assert(position(plug, W))].

rule f13:
[1: position(table_lamp, W),
 2: not(position(plug, W))]
==>
[assert(buy(extension_cord, W)),
  assert(position(plug, W))].

```

The program also uses rules to control a dialog with the user to gather initial data. It needs to know about room dimensions, furniture to be placed, plug locations, etc. It does this by setting various data gathering goals. For example the initial goal of the system is to **place_furniture**. It gives directions to the user and sets up the goal **read_furniture**. Once **read_furniture** is done (signified by the user entering end : end), it sets up the next goal of **read_walls**. Here is the beginning code.

```

rule 1:
[1: goal(place_furniture), % The initial goal causes a
 2: legal_furniture(LF)] % rule to fire with introductory info.
==> % It will set a new goal.
[retract(1),
  cls, nl,
  write('Enter a single item of furniture at each prompt. '), nl,
  write('Include the width (in feet) of each item. '), nl,
  write('The format is Item:Length. '), nl, nl,
  write('The legal values are: '), nl,
  write(LF), nl, nl,
  write('When there is no more furniture, enter "end:end".'), nl,
  assert(goal(read_furniture))].

rule 2:
[1: furniture(end, end), % When the furniture is read
 2: goal(read_furniture)] % set the new goal of reading
==> % wall sizes
[retract(all),
  assert(goal(read_walls))].

rule 3:
[1: goal(read_furniture), % Loop to read furniture.
 2: legal_furniture(LF)]
==>
[prompt('furniture> ', F:L),
  member(F, LF),
  assert(furniture(F, L))].

```

```

rule 4:                                % If rule 3 matched and failed
[1: goal(read_furniture),              % the action, then member
 2: legal_furniture(LF)]               % must have failed.
==>
[write('Unknown piece of furniture, must be one of:'), nl,
 write(LF), nl].

```

The room configurer illustrates both the strengths and weaknesses of a pure production system. The rules for laying out the furniture are very clear. New rules can be added, and old ones deleted without affecting the system. It is much easier to work with this program structure than it would be to understand procedural code that attempted to do the same thing.

On the other hand, the rules which interact with the user to collect data are difficult to read and have interdependencies, which make them hard to maintain. The flow of control is obscured. This code would be better written procedurally, but it is done using *Oops* to illustrate how these kinds of problems can be solved with a production architecture.

An ideal architecture would integrate the two approaches. It would be very simple to let *Oops* drop back down to Prolog for those cases where *Oops* is inappropriate. This architecture is covered in chapter 7.

5.3 Implementation

The implementation of *Oops* is both compact and readable due to the following features of Prolog:

- Each rule is represented as a single Prolog term (a relatively complex structure).
- The functors of the rule structure are defined as operators to allow the easy-to-read syntax of the rule.
- Prolog's built-in backtracking search makes rule selection easy.
- Prolog's built-in pattern matching (unification) makes comparison with working storage easy.
- Since each rule is a single term, unification causes variables to be automatically bound between LHS conditions and RHS actions.
- The Prolog database provides an easy representation of working storage.

Each rule is a single Prolog term, composed primarily of two lists: the right hand side (RHS), and the left hand side (LHS). These are stored using Prolog's normal data structures, with **rule** being the predicate and the various arguments being lists.

In *Clam*, DCG was used to allow a friendly, flexible rule format. In *Oops*, Prolog operators are used. The operators allow for a syntax that is formal, but readable. The operator syntax can also be used directly in the code.

Without operator definitions, the rules would look like normal hierarchical Prolog data structures:

```

rule(==>(:(id4, [:(1, flies(X)), :(2, lays_eggs(X))],
[retract(all), assert(isa(X, bird))])).

```

The following operator definitions allow for the more readable format of the rules:

```

op(230, xfx, ==>).
op(32, xfy, :).

```

```
op(250, fx, rule).
```

Working storage is stored in the database under the key **fact**. So to add a term to working storage:

```
asserta( fact(isa(robie, carnivore)) ),
```

and to look for a term in working storage:

```
fact( isa(X, carnivore) ).
```

Figure 5.1 shows the major predicates in the *Oops* inference engine. The inference cycle of recognize-act is coded in the predicate **go**. It searches for the first rule which matches working storage, and executes it. Then it repeats the process. If no rules match, then the second clause of **go** is executed and the inference ends. Then the second clause prints out the current state showing what was determined during the run. (Note that **LHS** and **RHS** are bound to lists.)

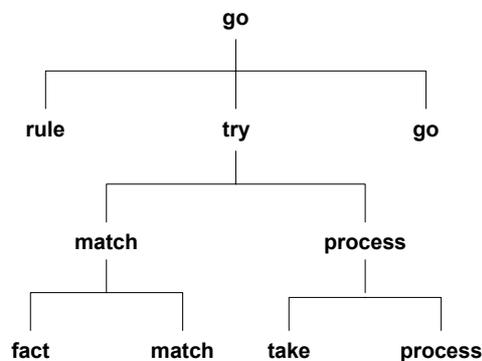


Figure 5.1 Major predicates in *Oops* inference engine

```
go:-
  call(rule ID: LHS ==> RHS),
  try(LHS, RHS),
  write('Rule fired '), write(ID), nl,
  !,
  go.
go:-
  nl, write(done), nl,
  print_state.
```

This code illustrates the tremendous expressiveness of Prolog. The code is very tight due to the built-in pattern matching and backtracking search. Especially note that since the entire rule is a single Prolog term, the unification between variables in the conditions and actions happens automatically. This leads to a use of variables that is, in some senses, cleaner and more powerful than that found in OPS5.

The **try/2** predicate is very simple. If **match/2** fails, it forces **go** to backtrack and find the next rule. The **LHS** is passed to **process** so retract statements can find the facts to retract.

```
try(LHS, RHS):-
  match(LHS),
  process(RHS, LHS).
```

The **match/2** predicate recursively goes through the list of conditions, locating them in working storage. If the condition is a comparison test, then the test is tried, rather than searched for in the database.

```
match([]) :- !.
match([N:Prem|Rest]) :-
    !,
    (fact(Prem);
     test(Prem)), % a comparison test rather than a fact
    match(Rest).
match([Prem|Rest]) :-
    (fact(Prem); % condition number not specified
     test(Prem)),
    match(Rest).
```

The **test/1** predicate can be expanded to include any kind of test. *Oops* uses most of the basic tests provided with Prolog, plus a few. For example:

```
test(X >= Y) :-
    X >= Y,
    !.
test(X = Y) :-
    X is Y, % use = for arithmetic
    !.
test(X # Y) :-
    X = Y, % use # for unification
    !.
test(member(X, Y)) :-
    member(X, Y),
    !.
test(not(X)) :-
    fact(X),
    !,
    fail.
```

If **match/2** succeeds, then **process/2** is called. It executes the RHS list of actions. It is equally straightforward.

```
process([], _) :- !.
process([Action|Rest], LHS) :-
    take(Action, LHS),
    process(Rest, LHS).
```

Only the action **retract** needs the **LHS**. The **take/2** predicate does a retract if that is what's called for, or else passes control to **take/1**, which enumerates the simpler actions. Some examples are given here.

```
take(retract(N), LHS) :-
    (N == all; integer(N)),
    retr(N, LHS),
    !.
take(A, _) :-
    take(A),
    !.
take(retract(X)) :-
    retract(fact(X)),
    !.
take(assert(X)) :-
    asserta(fact(X)),
    write(adding-X), nl,
    !.
take(X # Y) :-
    X = Y,
    !.
```

```

take(X = Y) :-
    X is Y,
    !.
take(write(X)) :-
    write(X),
    !.
take(nl) :-
    nl,
    !.
take(read(X)) :-
    read(X),
    !.

```

The **retr** predicate searches for **LHS** conditions with the same identification (**N**) and retracts them. If **all** was indicated, then it retracts all of the **LHS** conditions.

```

retr(all, LHS) :-
    retrall(LHS),
    !.
retr(N, []) :-
    write('retract error, no '-N), nl,
    !.
retr(N, [N:Prem|_]) :-
    retract(fact(Prem)),
    !.
retr(N, [_|Rest]) :-
    !,
    retr(N, Rest).

retrall([]).
retrall([N:Prem|Rest]) :-
    retract(fact(Prem)),
    !,
    retrall(Rest).
retrall([Prem|Rest]) :-
    retract(fact(Prem)),
    !,
    retrall(Rest).
retrall([_|Rest]) :- % must have been a test
    retrall(Rest).

```

Oops can be made to look like the other shells by the addition of a command loop predicate with commands similar to those in *Clam* and *Native*. Figure 5.2 shows the architecture of *Oops*.

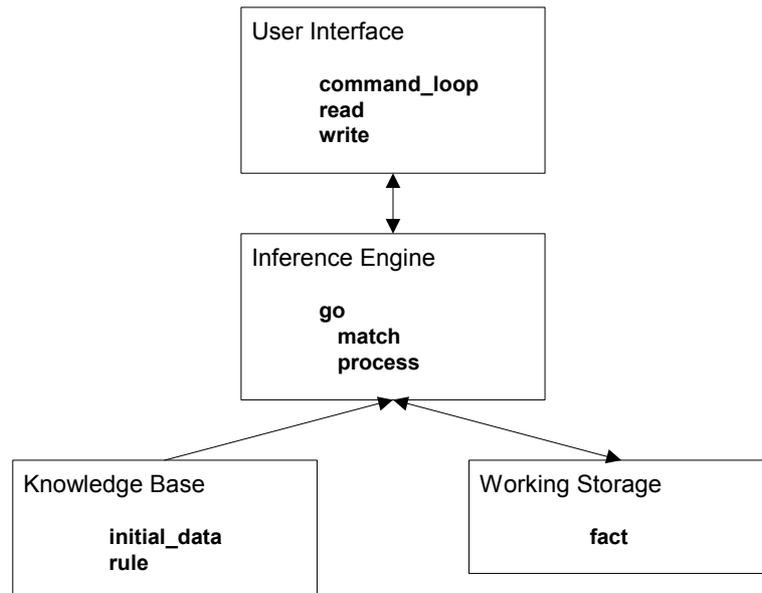


Figure 5.2 Major predicates of the Oops shell

5.4 Explanations for Oops

Explanations for forward chaining systems are more difficult to implement. This is because each rule modifies working storage, thus covering its tracks. The most useful information in debugging a forward chaining system is a trace facility. That is, you want to know each rule that is fired and the effects it has on working storage.

Each fact can have associated with it the rule which posted it, and this would give the immediate explanation of a fact. However, the facts that supported the rules which led up to that fact might have been erased from working memory. To give a full explanation, the system would have to keep time stamped copies of old versions of facts.

The trace option is added in *Oops* in a similar fashion to which it was added in *Clam*. The inference engine informs the user of the rules that are firing as they fire.

5.5 Enhancements

Oops in its current state is a simple forward chaining system. More advanced forward chaining systems differ in two main aspects.

- more sophisticated rule selection when many rules match the current working storage;
- performance.

The current rule selection strategy of *Oops* is simply to pick the first rule that matches. If many rules match, there might be other optimal choosing strategies. For example, we could pick the rule that matched the most recently asserted facts, or the rule that had the most specific match. Either of these would change the inference pattern of the system to give effects that might be more natural.

Oops is also inefficient in its pattern matching, since at each cycle of the system it tries all of the rules against working memory. There are various indexing schemes that can be used to allow for much faster picking of rules which match working memory. These will be discussed in chapter 8.

5.6 Rule Selection

OPPS5, which is probably the most well known example of a forward chaining, or production, system offers two different means of selecting rules. One is called LEX and the other is MEA. Both make use of time stamped data to determine the best rule to fire next. They differ slightly in the way in which they use the data. Both of these strategies can be added to *Oops* as options.

Generating the conflict set

For both, the first step is to collect all of the rules whose LHS match working memory at a given cycle. This set of rules is called the conflict set. It is not actually the rules, but rather instantiations of the rules. This means that the same rule might have multiple instantiations if there are multiple facts which match a LHS premise. This will often happen when there are variables in the rules that are bound differently for different instantiations.

For example, an expert system to identify animals might have the following condition on the LHS:

```
rule 12:
[...
 eats(X, meat),
 ...]
==> ...
```

In working memory there might be the following two facts:

```
...
eats(robie, meat).
eats(suzie, meat).
...
```

Assuming the other conditions on the LHS matched, this would lead to two different instantiations of the same rule. One for **robie** and one for **suzie**.

The simplest way to get the conflict set is to use **findall** or its equivalent. (If your system does not have a **findall**, a description of how to write your own can be found in Clocksin and Mellish section 7.8, *Assert and Retract: Random, Gensym, Findall*.) It collects all of the instantiations of a term in a list. The three arguments to **findall** are:

- a term that is used as a pattern to collect instantiations of variables;
- a list of goals used as a query;
- an output list whose elements match the pattern of the first argument, and for which there is one element for each successful execution of the query in the second argument.

The instantiations of the conflict set will be stored in a structure, **r/4**. The last three arguments of **r/4** will be the **ID**, **LHS**, and **RHS** of the rule, which will be used later.

The first argument of **r/4** is the **LHS** with the variables instantiated with the working storage elements that were matched. Each match of a LHS premise and working storage element is also accompanied by a time stamp indicating when the working storage element was last updated.

The query to be executed repeatedly by **findall** will be similar to the one currently used to find just the first matching rule:

```
?- rule ID : LHS ==> RHS, match(LHS, Inst)
```

Note that **match** now has a second argument to store the instantiation of the rule, which will be the first argument to **r/4**.

The following predicate puts the above pieces together and uses **findall** to build a list (**CS**) of **r/4**s representing all of the rules that currently match working storage.

```
conflict_set(CS) :-
    findall(r(Inst, ID, LHS, RHS),
           [rule ID: LHS ==> RHS, match(LHS, Inst)],
           CS).
```

The **match** predicate just needs to be changed to capture the instantiation of the conditions. Notice there is an additional argument, **Time**, in the **fact** predicate. This is a time stamp that will be used in the selection process.

```
match([], []) :- !.
match([N:Prem|Rest], [Prem/Time|IRest]) :-
    !,
    (fact(Prem, Time);
     test(Prem), Time = 0), %comparison, not a fact
    match(Rest, IRest).
match([Prem|Rest], [Prem/Time|IRest]) :-
    (fact(Prem, Time); % no condition number
     test(Prem), Time = 0),
    match(Rest, IRest).
```

Time stamps

The timestamp is just a chronological counter that numbers the **facts** in working memory as they are added. All assertions of facts are now handled by the **assert_ws** predicate as follows:

```
assert_ws(fact(X, T)) :-
    getchron(T),
    asserta(fact(X, T)).
```

The **getchron** predicate simply keeps adding to a counter:

```
getchron(N) :-
    retract( chron(N) ),
    NN is N + 1,
    asserta( chron(NN) ),
    !.
```

5.7 LEX

Now that we have a list of possible rules and instantiations in the conflict set, it is necessary to select one. First we will look at the OPS5 LEX method of rule selection. It uses three criteria to select a rule.

The first is refraction. This discards any instantiations that have already been fired. Two instantiations are the same if the variable bindings and the time stamps are the same. This prevents the same rule from firing over and over — unless the programmer has caused working memory to be repeatedly updated with the same fact.

The second is recency. This chooses the rules that use the most recent elements in working memory. The conflict set rules are ordered with those rules with the highest time stamps first. This is useful to give the system a sense of focus as it works on a problem. Facts that are just asserted will most likely be used next in a rule.

The third is specificity. If there are still multiple rules in contention, the most specific is used. The more conditions there are that apply in the LHS of the rule, the more specific it is. For example, a rule with 4 conditions is more specific than one with 3 conditions. This criteria ensures that general case rules will fire after more specific case rules.

If after these tests there are still multiple rules which apply, then one is chosen at random.

Changes in the Rules

The LEX strategy changes the way in which *Ops* rules are programmed. In the first version of *Ops*, the knowledge engineer had to make sure that the working storage elements that caused the rule to fire are changed. It was the knowledge engineer's responsibility to ensure that a rule did not repeatedly fire.

The opposite is also true. Where looping is required, the facts matching the LHS must be continually reasserted.

In the original version of *Ops* the knowledge engineer knew the order in which rules would fire, and could use that information to control the inference. Using LEX he/she can still control the inference, but it requires more work. For example, if it is desirable to have the couch placed first by the system, then that rule must be structured to fire first. This can be done by adding a goal to place the couch first and asserting it after the data is gathered. For example:

```
rule gather_data
...
==>
[...
  assert( goal(couch_first) ) ].
rule couch
[ goal(couch_first),
...

```

The **gather_data** rule will assert the **couch_first** goal after all other assertions. This means it is the most recent addition to working storage. The Lex recency criteria will then ensure that the couch rule is fired next.

The rule which is supposed to fire last in the system also needs to be handled specially. The easiest way to ensure a rule will fire last is to give it an empty list for the LHS. The specificity check will keep it from firing until all others have fired.

Implementing LEX

To implement the LEX strategy, we modify the **go** predicate to first get the conflict set and then pass it to the predicate **select_rule**, which picks the rule to execute. After processing the rule, the instantiation associated with the rule is saved to be used as a check that it is not re-executed.

```
go :-
  conflict_set(CS),
  select_rule(CS, r(Inst, ID, LHS, RHS)),
  process(RHS, LHS),
  asserta( instantiation(Inst) ),
```

```

    write('Rule fired '), write(ID), nl,
    !,
    go.
go.

```

The **select_rule** predicate applies the three criteria above to select the appropriate rule. The **refract** predicate applies refraction, and **lex_sort** applies both recency and specificity through a sorting mechanism.

```

select_rule(CS, R) :-
    refract(CS, CS1),
    lex_sort(CSR, [R|_]).

```

First let's look at **refract**, which removes those rules that duplicate existing instantiations. It relies on the fact that after each successful rule firing, the instantiation is saved in the database.

```

refract([], []).
refract([r(Inst, _, _, _) | T], TR) :-
    instantiation(Inst),
    !,
    refract(T, TR).
refract([H|T], [H|TR]) :-
    refract(T, TR).

```

Once **refract** is done processing the list, only those rules with new instantiations are left on the list.

The implementation of **lex_sort** doesn't filter the remaining rules, but sorts them so that the first rule on the list is the desired rule. This is done by creating a key for each rule that is used to sort the rules. The key is designed to sort by recency and specificity. The sorting is done with a common built-in predicate, **keysort**, which sorts a list by keys where the elements are in the form: **key – term**. (If your Prolog does not have a **keysort**, see Clocksin and Mellish section 7.7, *Sorting*.)

```

lex_sort(L, R) :-
    build_keys(L, LK),
    keysort(LK, X),
    reverse(X, Y),
    strip_keys(Y, R).

```

The **build_keys** predicate adds the keys to each term. The keyed list is then sorted by **keysort**. It comes out backwards, so it is reversed, and finally the keys are stripped from the list. In order for this to work, the right key needs to be chosen.

The key that gives the desired results is itself a list. The elements are the time stamps of the various matched conditions in the instantiation of the rule. The key (a list) is sorted so that the most recent (highest number) time stamps are at the head of the list. These complex keys can themselves be sorted to give the correct ordering of the rules. For example, consider the following two rules, and working storage:

```

rule t1:
    [flies(X),
     lays_eggs(X)]
==>
    [assert(bird(X))].

rule t2:
    [mammal(X),
     long_ears(X),
     eats_carrots(X)]

```

```

==>
[assert(animal(X, rabbit))].
fact( flies(lara), 9).
fact( flies(zach), 6).
fact( lays_eggs(lara), 7).
fact( lays_eggs(zach), 8).
fact( mammal(bonbon), 3).
fact( long_ears(bonbon), 4).
fact( eats_carrots(bonbon), 5).

```

There would be two instantiations of the first rule, one each for **lara** and **zach**, and one instantiation of the second rule for **bonbon**. The highest numbers are the most recent time stamps. The keys (in order) for these three instantiations would be:

```

[9, 7]
[8, 6]
[5, 4, 3]

```

In order to get the desired sort, lists must be compared element by element starting from the head of the list. This gives the **recency** sort. The sort must also distinguish between two lists of different lengths with the same common elements. This gives the **specificity** sort. For AAIS prolog the sort works as desired with **recency** being more important than **specificity**. It should be checked for other Prologs.

Here is the code to build the keys:

```

build_keys([], []).
build_keys([r(Inst, A, B, C)|T], [Key-r(Inst, A, B, C)|TR]) :-
    build_chlist(Inst, ChL),
    sort(ChL, X),
    reverse(X, Key),
    build_keys(T, TR).

build_chlist([], []).
build_chlist([_ /Chron|T], [Chron|TC]) :-
    build_chlist(T, TC).

```

The **build_keys** predicate uses **build_chlist** to create a list of the time stamps associated with the LHS instantiation. It then sorts those, and reverses the result, so that the most recent time stamps are first in the list.

The final predicate, **strip_keys**, simply removes the keys from the resulting list.

```

strip_keys([], []).
strip_keys([Key-X|Y], [X|Z]) :-
    strip_keys(Y, Z).

```

5.8 MEA

The other strategy offered with OPS5 is MEA. This is identical to LEX with one additional filter added. After refraction, it finds the time stamp associated with the first condition of the rule and picks the rule with the highest time stamp on the first condition. If there is more than one, then the normal LEX algorithm is used to pick which of them to use.

At first this might seem like an arbitrary decision; however, it was designed to make goal directed programming easier in OPS5. The flow of control of a forward chaining system is often controlled by setting goal facts in working storage. Rules might have goals in the conditions thus ensuring the rule will only fire when that goal is being pursued.

By making the goal condition the first condition on the LHS of each rule, and by using MEA, the programmer can force the system to pursue goals in a specified manner. In fact, using this technique it is possible to build backward chaining systems using a forward chaining tool.

The test for MEA is simply added to the system. First, the filter is added to the **select_rule** predicate. It will simply return the same conflict set if the current strategy is not MEA.

```
select_rule(R, CS) :-
    refract(CS, CS1),
    mea_filter(0, CS1, [], CSR),
    lex_sort(CSR, [R|_]).
```

The actual filter predicates build the new list in an accumulator variable, **Temp**. If the first time stamp for a given rule is less than the current maximum, it is not included. If it equals the current maximum, it is added to the list of rules. If it is greater than the maximum, that timestamp becomes the new maximum, and the list is reinitialized to have just that single rule.

```
mea_filter(_, X, _, X) :-
    not strategy(mea),
    !.
mea_filter(_, [], X, X).
mea_filter(Max, [r([A/T|Z], B, C, D)|X], Temp, ML) :-
    T < Max,
    !,
    mea_filter(Max, X, Temp, ML).
mea_filter(Max, [r([A/T|Z], B, C, D)|X], Temp, ML) :-
    T = Max,
    !,
    mea_filter(Max, X, [r([A/T|Z], B, C, D)|Temp], ML).
mea_filter(Max, [r([A/T|Z], B, C, D)|X], Temp, ML) :-
    T > Max,
    !,
    mea_filter(T, X, [r([A/T|Z], B, C, D)], ML).
```

These examples illustrate some of the difficulties with expert systems in general. The OPS5 programmer must be intimately familiar with the nature of the inferencing in order to get the performance desired from the system. He is only free to use the tools available to him.

On the other hand, if the programmer has access to the selection strategy code, and knows the type of inferencing that will be required, the appropriate strategy can be built into the system. Given the accessibility of the above code, it is easy to experiment with different selection strategies.

Exercises

- 5.1 Add full rule tracing to OOPS.
- 5.2 Add a command loop that turns on and off tracing, MEA/LEX strategies, loads rule files, consults the rules, lists working storage, etc.
- 5.3 Add a feature that allows for the saving of test case data, which can then be run against the system. The test data and the results are used to debug the system as it undergoes change.
- 5.4 Allow each rule to optionally have a priority associated with it. Use the user-defined rule priorities as the first criteria for selecting rule instantiations from the conflict set.

- 5.5 Add features on the LHS and RHS that allow rules to be written, which can access the conflict set and dynamically change the rule priorities. Figure out an application for this.
- 5.6 Add new syntax to the knowledge base that allows rules to be clustered into rule sets. Maintain separate conflict sets for each rule set and have the inference engine process each rule set to completion. Have higher level rules, which can be used to decide which rule sets to execute. Alternatively, each rule set can have an enabling pattern associated with it that allows it to fire just as individual rules are fired.
- 5.7 Each fact in working storage can be thought of as being dependent on other facts. The other facts are those that instantiated the LHS of a rule which updated the fact. By keeping track of these dependencies, a form of truth maintenance can be added to the system. When a fact is then removed from working storage, the system can find other facts that were dependent on it and remove them as well.

6 Frames

Up until this point in the book, we have worked with data structures that are simply that — data structures. It is often desirable to add some "intelligence" to the data structures, such as default values, calculated values, and relationships between data.

Of the various schemes which evolved over the years, the **frame** based approach has been one of the more popular. Information about an object in the system is stored in a *frame*. The frame has multiple *slots* used to define the various attributes of the object. The slots can have multiple *facets* for holding the value for the attributes, defaults or procedures that are called to calculate the value.

The various frames are linked together in a hierarchy with a-kind-of (ako) links that allow for inheritance. For example, rabbits and hamsters might be stored in frames that have ako(mammal). In the frame for mammal are all of the standard attribute-values for mammals, such as skin-fur and birth-live. These are inherited by rabbits and hamsters and do not have to be specified in their frames. There can also be defaults for attributes which might be overwritten by specific species. Legs-4 applies to most mammals, but monkeys would have legs-2 specified.

Another feature of a frame based system is *demons*. These are procedures that are activated by various updating procedures. For example, a financial application might have demons on various account balances that are triggered when the value is too low. These could also have editing capabilities that made sure the data being entered is consistent with existing data. Figure 6.1 shows some samples of frames for animals.

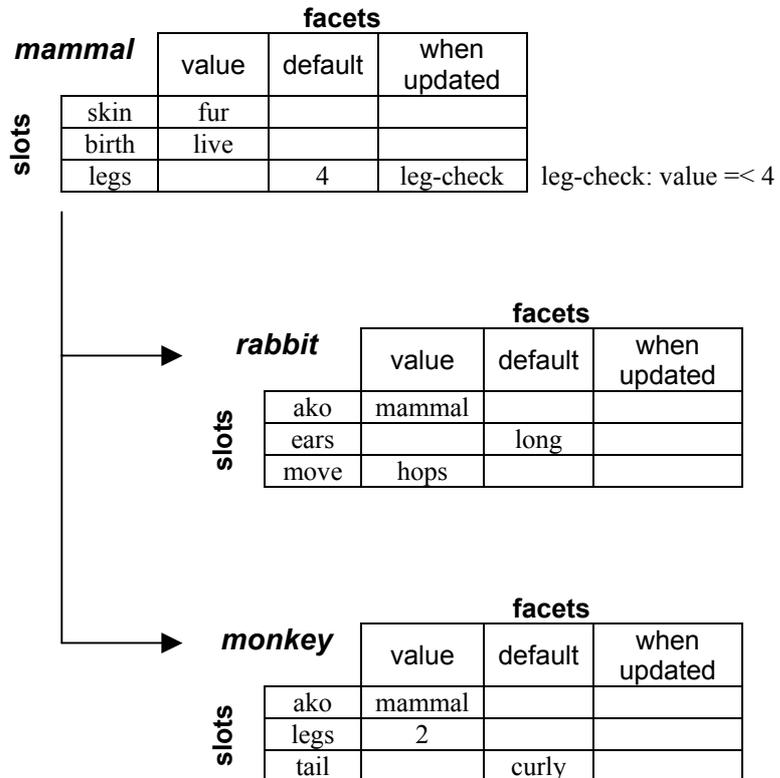


Figure 6.1 Examples of animal frames

6.1 The Code

In order to implement a frame system in Prolog, there are two initial decisions to be made. The first is the design of the user interface, and the second is the design of the internal data structure used to hold the frame information.

The access to data in the frame system will be through three predicates:

- **get_frame** — retrieves attribute values for a frame;
- **add_frame** — adds or updates attribute values for a frame;
- **del_frame** — deletes attribute values from a frame.

From the user's perspective, these operations will appear to be acting on structures that are very similar to database records. Each frame is like a record, and the slots in the frame correspond to the fields in the record. The intelligence in the frame system, such as inheritance, defaults, and demons, happens automatically for the user.

The first argument of each of these predicates is the name of the frame. The second argument has a list of the slots requested. Each slot is represented by a term of the form **attribute-value**. For example, to retrieve values for the **height** and **weight** slots in the frame **dennis**, the following query would be used:

```
?- get_frame(dennis, [weight-W, height-H]).
W = 155
H = 5-10
```

To add a new sport for **mary**:

```
?- add_frame(mary, [sport-rugby]).
```

To delete a slot's value for **mynorca's** computer:

```
?- del_frame(mynorca, [computer-'PC AT']).
```

These three primitive frame access predicates can be used to build more complex frame applications. For example, the following query would find all of the women in the frame database who are rugby players:

```
?- get_frame(X, [ako-woman, sport-rugby]).
X = mary ;
X = kelly;
```

A match-making system might have a predicate that looks for men and women who have a hobby in common:

```
in_common(M, W, H) :-
    get_frame(M, [ako-man, hobby-H]),
    get_frame(W, [ako-woman, hobby-H]).
```

6.2 Data Structure

The next decision is to choose a type of data structure for the frames. The frame is a relatively complex structure. It has a name and multiple slots. Each slot, which corresponds to an attribute of the frame, can have a value. This is the same as in a normal database. However in a frame system, the value is just one possible facet for the slot.

There might also be default values, predicates which are used to calculate values, and demons which fire when the value in the slot is updated.

Furthermore, the frames can be organized in a hierarchy, where each frame has an a-kind-of slot that has a list of the types of frames from which this frame inherits values.

The data structure chosen for this implementation has the predicate name **frame** with two arguments. The first is the name of the frame, and the second is a list of slots separated by a hyphen operator from their respective facet lists. The facet list is composed of prefix operator facet names. The ones defined in the system are:

- **val** — the simple value of the slot;
- **def** — the default if no value is given;
- **calc** — the predicate to call to calculate a value for the slot;
- **add** — the predicate to call when a value is added for the slot;
- **del** — the predicate to call when the slot's value is deleted.

Here is the format of a frame data structure:

```
frame(name, [
    slotname1 - [
        facet1 val11,
        facet2 val12,
        ...],
    slotname2 - [
        facet1 val21,
        facet2 val 22,
        ...],
    ...]).
```

For example:

```
frame(man, [
    ako-[val [person]],
    hair-[def short, del bald],
    weight-[calc male_weight] ]).
frame(woman, [
    ako-[val [person]],
    hair-[def long],
    weight-[calc female_weight] ]).
```

In this case, both **man** and **woman** have **ako** slots with the value of **person**. The hair slot has default values of long and short hair for women and men, but this would be overridden by the values in individual frames. Both have facets that point to predicates that are to be used to calculate weight, if none is given. The man's hair slot has a facet that points to a demon, **bald**, to be called if the value for hair is deleted.

One additional feature permits values to be either single-valued or multi-valued. Single values are represented by terms, multiple values are stored in lists. The **add_frame** and **del_frame** predicates take this into account when updating the frame. For example **hair** has a single value but **hobbies** and **ako** can have multiple values.

6.3 The Manipulation Predicates

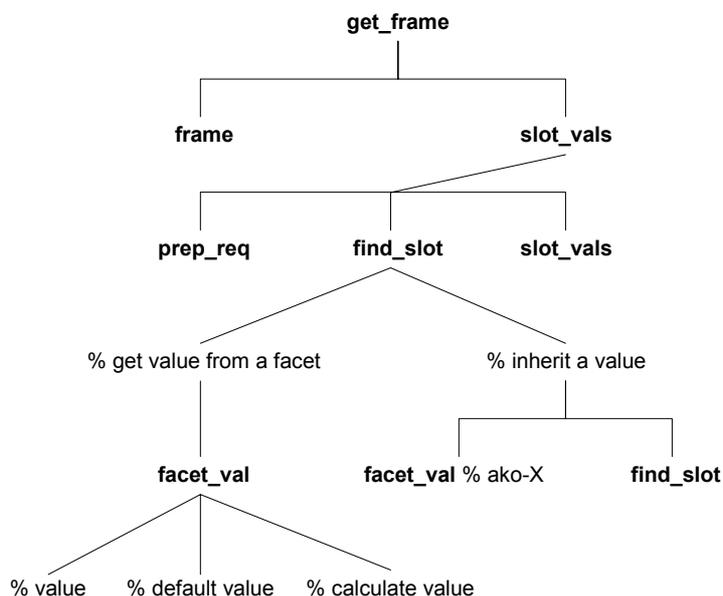


Figure 6.2 Major predicates of `get_frame`

The first predicate to look at is `get_frame`. It takes as input a query pattern, which is a list of slots and requested values. This request list (**ReqList**) is then compared against the **SlotList** associated with the frame. As each request is compared against the slot list, Prolog's unification instantiates the variables in the list. Figure 6.2 shows the major predicates used with `get_frame`.

```

get_frame(Thing, ReqList) :-
    frame(Thing, SlotList),
    slot_vals(Thing, ReqList, SlotList).
  
```

The `slot_vals` predicate takes care of matching the request list against the slot list. It is a standard recursive list predicate, dealing with one item from the request list at a time. That item is first converted from the more free-form style allowed in the input list to a more formal structure describing the request. That structure is **req/4** where the arguments are:

- name of the frame;
- the requested slot;
- the requested facet;
- the requested value.

The code for `slot_vals` recognizes request lists, and also single slot requests not in list form. This means both of the following frame queries are legal:

```

?- get_frame( dennis, hair-X ).
...
?- get_frame( dennis, [hair-X, height-Y] ).
...
  
```

The **slot_vals** predicate is a standard list recursion predicate that fulfills each request on the list in turn. The real work is done by **find_slot**, which fulfills the request from the frame's slot list.

```
slot_vals(_, [], _).
slot_vals(T, [Req|Rest], SlotList) :-
    prep_req(Req, req(T, S, F, V)),
    find_slot(req(T, S, F, V), SlotList),
    !,
    slot_vals(T, Rest, SlotList).
slot_vals(T, Req, SlotList) :-
    prep_req(Req, req(T, S, F, V)),
    find_slot(req(T, S, F, V), SlotList).
```

The request list is composed of items of the form **Slot - X**. The **prep_req** predicate, which builds the formal query structure, must recognize three cases:

- **X** is a variable, in which case the value of the slot is being sought.
- **X** is of the form **Facet(Val)**, in which case a particular facet is being sought.
- **X** is a non-variable, in which case the slot value is being sought for comparison with the given value.

Here is the code which prepares the formal query structure:

```
prep_req(Slot-X, req(T, Slot, val, X)) :-
    var(X),
    !.
prep_req(Slot-X, req(T, Slot, Facet, Val)) :-
    nonvar(X),
    X =.. [Facet, Val],
    facet_list(FL),
    member(Facet, FL),
    !.
prep_req(Slot-X, req(T, Slot, val, X)).
    facet_list([val, def, calc, add, del, edit]).
```

For example, the query

```
?- get_frame(dennis, [hair-X]).
```

would generate the more formal request for:

```
req(dennis, hair, val, X)
```

Having now prepared a more formal request, and a slot list to fulfill it, **find_slot** attempts to satisfy the request. The first clause handles the case where the request is not for a variable, but really just a test to see if a certain value exists. In this case another request with a different variable (**Val**) is started, and the results compared with the original request. Two cases are recognized: either the value was a single value, or the value was a member of a list of values.

```
find_slot(req(T, S, F, V), SlotList) :-
    nonvar(V),
    find_slot(req(T, S, F, Val), SlotList),
    !,
    (Val == V;
    member(V, Val)).
```

The next clause covers the most common case, in which the value is a variable, and the slot is a member of the slot list. Notice that the call to **member** both verifies that there is a structure of the form **Slot-FacetList** and unifies **FacetList** with the list of facets associated with the slot. This is because **S** is bound at the start of the call to **member**, and **FacetList** is not. Next, **facet_val** is called, which gets the value from the facet list.

```
find_slot(req(T, S, F, V), SlotList) :-
    member(S-FacetList, SlotList),
    !,
    facet_val(req(T, S, F, V), FacetList).
```

If the requested slot was not in the given slot list for the frame, then the next clause uses the values in the **ako** (a-kind-of) slot to see if there is a super class from which to inherit a value. The value in the **ako** slot might be a list, or a single value. The higher frame's slot list is then used in an attempt to satisfy the request. Note that this recurses up through the hierarchy. Note also that a frame may have multiple values in the **ako** slot, allowing for a more complex structure than a pure hierarchy. The system works through the list in order, trying to satisfy a request from the first **ako** value first.

```
find_slot(req(T, S, F, V), SlotList) :-
    member(ako-FacetList, SlotList),
    facet_val(req(T, ako, val, Ako), FacetList),
    (member(X, Ako);
     X = Ako),
    frame(X, HigherSlots),
    find_slot(req(T, S, F, V), HigherSlots),
    !.
```

The final clause in **find_slot** calls the error handling routine. The error handling routine should probably be set not to put up error messages in general, since often quiet failure is required. During debugging it is useful to have it turned on.

```
find_slot(Req, _) :-
    error(['frame error looking for:', Req]).
```

The **facet_val** predicate is responsible for getting the value for the facet. It deals with four possible cases:

- The requested facet and value are on the facet list. This covers the **val** facet as well as specific requests for other facets.
- The requested facet is **val**, it is on the facet list, and its value is a list. In this case **member** is used to get a value.
- There is a default (**def**) facet, which is used to get the value.
- There is a predicate to call (**calc**) to get the value. It expects the formal request as an argument.

If the facet has a direct value in the facet list, then there is no problem. If there is not a direct value, and the facet being asked for is the **val** facet, then alternate ways of getting the value are used. First the default is tried and, if there is no default, a **calc** predicate is used to compute the value. If a **calc** predicate is needed, then the call to it is built using the **univ** (=..) built-in predicate, with the request pattern as the first argument, and other arguments included in the **calc** predicate following.

```
facet_val(req(T, S, F, V), FacetList) :-
    FV =.. [F, V],
    member(FV, FacetList),
    !.
```

```

facet_val(req(T, S, val, V), FacetList) :-
    member(val ValList, FacetList),
    member(V, ValList),
    !.
facet_val(req(T, S, val, V), FacetList) :-
    member(def V, FacetList),
    !.
facet_val(req(T, S, val, V), FacetList) :-
    member(calc Pred, FacetList),
    Pred =.. [Functor | Args],
    CalcPred =.. [Functor, req(T, S, val, V) | Args],
    call(CalcPred).

```

An example of a predicate used to calculate values is the **female_weight** predicate. It computes a default weight equal to twice the height of the individual.

```

female_weight(req(T, S, F, V)) :-
    get_frame(T, [height-H]),
    V is H * 2.

```

We have now seen the code that gets values from a frame. It first sets up a list of requested slot values and then processes them one at a time. For each slot which is not defined for the frame, inheritance is used to find a parent frame that defines the slot. For the slots that are defined, each of the facets is tried in order to determine a value.

The next major predicate in the frame system adds values to slots. For single valued slots, this is a replace. For multi-valued slots, the new value is added to the list of values.

The **add_frame** predicate uses the same basic form as **get_frame**. For updates, first the old slot list is retrieved from the existing frame. Then the predicate **add_slots** is called with the old list (**SlotList**) and the update list (**UList**). It returns the new list (**NewList**).

```

add_frame(Thing, UList) :-
    old_slots(Thing, SlotList),
    add_slots(Thing, UList, SlotList, NewList),
    retract(frame(Thing, _)),
    asserta(frame(Thing, NewList)),
    !.

```

The **old_slots** predicate usually just retrieves the slot list. However, if the frame doesn't exist, it creates a new frame with an empty slot list.

```

old_slots(Thing, SlotList) :-
    frame(Thing, SlotList),
    !.
old_slots(Thing, []) :-
    asserta(frame(Thing, [])).

```

Next, comes **add_slots**, which does analogous list matching to **slot_vals** called by **get_frame**.

```

add_slots(_, [], X, X).
add_slots(T, [U|Rest], SlotList, NewList) :-
    prep_req(U, req(T, S, F, V)),
    add_slot(req(T, S, F, V), SlotList, Z),
    add_slots(T, Rest, Z, NewList).
add_slots(T, X, SlotList, NewList) :-
    prep_req(X, req(T, S, F, V)),
    add_slot(req(T, S, F, V), SlotList, NewList).

```

The **add_slot** predicate deletes the old slot and associated facet list from the old slot list. It then adds the new facet and value to that facet list and rebuilds the slot list. Note that

delete unifies **FacetList** with the old facet list. **FL2** is the new facet list returned from **add_facet**. The new slot and facet list, **S-FL2** is then made the head of the **add_slot** output list, with **SL2**, the slot list after deleting the old slot as the tail.

```
add_slot(req(T, S, F, V), SlotList, [S-FL2|SL2]) :-
    delete(S-FacetList, SlotList, SL2),
    add_facet(req(T, S, F, V), FacetList, FL2).
```

The **add_facet** predicate takes the request and deletes the old facet from the list, builds a new facet and adds it to the facet list in the same manner as **add_slot**. The main trickiness is that **add_facet** makes a distinction between a facet whose value is a list, and one whose value is a term. In the case of a list, the new value is added to the list, whereas in the case of a term, the old value is replaced. The **add_newval** predicate does this work, taking the **OldVal**, the new value **V** and coming up with the **NewVal**.

```
add_facet(req(T, S, F, V), FacetList, [FNew|FL2]) :-
    FX =.. [F, OldVal],
    delete(FX, FacetList, FL2),
    add_newval(OldVal, V, NewVal),
    !,
    check_add_demons(req(T, S, F, V), FacetList),
    FNew =.. [F, NewVal].

add_newval(X, Val, Val) :-
    var(X),
    !.
add_newval(OldList, ValList, NewList) :-
    list(OldList),
    list(ValList),
    append(ValList, OldList, NewList),
    !.
add_newval([H|T], Val, [Val, H|T]).
add_newval(Val, [H|T], [Val, H|T]).
add_newval(_, Val, Val).
```

The intelligence in the frame comes after the cut in **add_facet**. If a new value has been successfully added, then **check_add_demons** looks for any demon procedures that must be run before the update is completed.

In **check_add_demons**, **get_frame** is called to retrieve any demon predicates in the facet **add**. Note that since **get_frame** uses inheritance, demons can be put in higher level frames that apply to all sub frames.

```
check_add_demons(req(T, S, F, V), FacetList) :-
    get_frame(T, S-add(Add)),
    !,
    Add =.. [Functor | Args],
    AddFunc =.. [Functor, req(T, S, F, V) | Args],
    call(AddFunc).
check_add_demons(_, _).
```

The **delete** predicate used in the add routines must simply return a list that does not have the item to be deleted in it. If there was no item, then returning the same list is the right thing to do. Therefor **delete** looks like:

```
delete(X, [], []).
delete(X, [X|Y], Y) :-
    !.
delete(X, [Y|Z], [Y|W]) :-
    delete(X, Z, W).
```

The **del_frame** predicate is similar to both **get_frame** and **add_frame**. However, one major difference is in the way items are deleted from lists. When **add_frame** was deleting things from lists (for later replacements with updated values), the behavior of delete above was appropriate. For **del_frame**, a failure should occur if there is nothing to delete from the list. For this function we use **remove**, which is similar to **delete**, but fails if there was nothing to delete.

```
remove(X, [X|Y], Y) :- !.
remove(X, [Y|Z], [Y|W]) :-
    remove(X, Z, W).
```

The rest of **del_frame** looks like:

```
del_frame(Thing) :-
    retract(frame(Thing, _)).
del_frame(Thing) :-
    error(['No frame', Thing, 'to delete']).
del_frame(Thing, UList) :-
    old_slots(Thing, SlotList),
    del_slots(Thing, UList, SlotList, NewList),
    retract(frame(Thing, _)),
    asserta(frame(Thing, NewList)).

del_slots([], X, X, _).
del_slots(T, [U|Rest], SlotList, NewList) :-
    prep_req(U, req(T, S, F, V)),
    del_slot(req(T, S, F, V), SlotList, Z),
    del_slots(T, Rest, Z, NewList).
del_slots(T, X, SlotList, NewList) :-
    prep_req(X, req(T, S, F, V)),
    del_slot(req(T, S, F, V), SlotList, NewList).

del_slot(req(T, S, F, V), SlotList, [S-FL2|SL2]) :-
    remove(S-FacetList, SlotList, SL2),
    del_facet(req(T, S, F, V), FacetList, FL2).
del_slot(Req, _, _) :-
    error(['del_slot - unable to remove', Req]).

del_facet(req(T, S, F, V), FacetList, FL) :-
    FV =.. [F, V],
    remove(FV, FacetList, FL),
    !,
    check_del_demons(req(T, S, F, V), FacetList).
del_facet(req(T, S, F, V), FacetList, [FNew|FL]) :-
    FX =.. [F, OldVal],
    remove(FX, FacetList, FL),
    remove(V, OldVal, NewValList),
    FNew =.. [F, NewValList],
    !,
    check_del_demons(req(T, S, F, V), FacetList).
del_facet(Req, _, _) :-
    error(['del_facet - unable to remove', Req]).

check_del_demons(req(T, S, F, V), FacetList) :-
    get_frame(T, S-del(Del)),
    !,
    Del =.. [Functor|Args],
    DelFunc =.. [Functor, req(T, S, F, V)|Args],
    call(DelFunc).
check_del_demons(_, _).
```

This code is essentially the same as for the **add** function, except that the new facet values have elements deleted instead of replaced. Also, the **del** facet is checked for demons instead of the **add** facet.

Here is an example of a demon called when a man's hair is deleted. It checks with the user before proceeding.

```
bald(req(T, S, F, V)) :-  
    write_line([T, 'will be bald, ok to proceed?']),  
    read(yes).
```

6.4 Using Frames

The use of inheritance makes the frame based system an intelligent way of storing data. For many expert systems, a large portion of the intelligence can be stored in frames instead of in rules. Let's consider, for example, the bird identification expert system.

In the bird system there is a hierarchy of information about birds. The rules about the order tubenose, family albatross, and particular albatrosses can all be expressed in frames as follows:

```
frame(tubenose,  
    [  
        level-[val order],  
        nostrils-[val external_tubular],  
        live-[val at_sea],  
        bill-[val hooked]  
    ]  
).  
frame(albatross,  
    [  
        ako-[val tubenose],  
        level-[val family],  
        size-[val large],  
        wings-[val long-narrow]  
    ]  
).  
frame(laysan_albatross,  
    [  
        ako-[val albatross],  
        level-[val species],  
        color-[val white]  
    ]  
).  
frame(black_footed_albatross,  
    [  
        ako-[val albatross],  
        level-[val species],  
        color-[val dark]  
    ]  
).
```

In a forward chaining system, we would feed some facts to the system and the system would identify the bird based on those facts. We can get the same behavior with the frame system and the predicate **get_frame**.

For example, if we know a bird has a dark color, and long narrow wings, we can ask the query:

```
?- get_frame(X, [color-dark, wings-long_narrow]).  
X = black_footed_albatross ;  
no
```

Notice that this will find all of the birds that have the requested property. The **ako** slots and inheritance will automatically apply the various slots from wherever in the hierarchy they appear. In the above example, the color attribute was filled from the black footed albatross frame and the wings attribute was filled from the albatross frame. This feature can be used to find all birds with long narrow wings:

```
?- get_frame(X, [wings-long_narrow]).
X = albatross ;
X = black_footed_albatross ;
X = laysan_albatross ;
no
```

The queries in this case are more general than in the various expert systems used so far. The query finds all frames that fit the given facts. The query could specify a level, but the query can also be used to bind variables for various fits. For example, to get the level in the hierarchy of the frames that have long narrow wings:

```
?- get_frame(X, [wings-long_narrow, level-L]).
X = albatross, L = family ;
X = black_footed_albatross, L = species;
X = laysan_albatross, L = species ;
no
```

6.5 Summary

For the expert systems we have seen already, we have used the Prolog database to store information. That database has been relatively simple. By writing special access predicates it is possible to create a much more sophisticated database using frame technology. These frames can then be used to store knowledge about the particular environment.

Exercises

- 6.1 Add other facets to the slots to allow for specification of things like explanation of the slot, certainty factors, and constraints.
- 6.2 Add an automatic query-the-user facility that is called whenever a slot value is sought and there is no other frame to provide the answer. This will allow the frame system to be used as a backward chaining expert system.

7 Integration

Many real problems cannot be solved by the simple application of a single expert system technique. Problems often require multiple knowledge representation techniques as well as conventional programming techniques.

Often it is necessary to either have the expert system embedded in a conventional application, or to have other applications callable from the expert system. For example, a financial expert system might need a tight integration with a spread sheet, or an engineering expert system might need access to programs that perform engineering calculations.

The degree to which the systems in this book can be integrated with other environments depends on the flexibility of the Prolog used and the application that needs to be accessed. The systems can be designed with the hooks in place for this integration. In the examples presented in this chapter, the knowledge base tools will have hooks to Prolog.

Often the Prolog code can be used to implement features of the system that do not fit neatly in the knowledge tools. This is often the case in the area of user interface, but applies to other portions as well. In the example in this chapter, we will see Prolog used to smooth over a number of the rough edges of the application.

The degree of integration needed between knowledge tools also depends somewhat on the application. In this chapter, the forward chaining system and frame based knowledge representation will be tightly integrated. By having control over the tools, the degree of integration can be implemented to suit the individual application.

The example used in this chapter is the same room furniture placement used in the chapter on forward chaining. While the application was developed with the pure *Oops* system, a much more elegant solution can be implemented by integrating frames, *Oops*, and Prolog. In particular, there is a lot of knowledge about the types of furniture that could be better stored in a frame system. Also the awkward input and output sections of the system could be better written in Prolog.

7.1 Foops (Frames and Oops)

The first step is to integrate the frame based knowledge representation with the *Oops* forward chaining inference engine. The integration occurs in two places:

- The concept of a frame instance needs to be implemented.
- The rules must be able to reference the frame instances.

The instances are needed for an integrated system to distinguish between the frame data definition in the knowledge base, and the instances of frames in working storage. The rules will be matching and manipulating instances of frames, rather than the frame definitions themselves. For example, there will be a frame describing the attributes of chairs, but there might be multiple instances of chairs in working storage.

Instances

In the frame system as it is currently written, the frames are the data. Particular instances of a frame, such as person, are just additional frames. For use in the expert system it is cleaner to distinguish between frame definitions and instances of frames.

The definitions specify the slots for a frame, and the instances provide specific values for the slots. The frame instances will be updated in working storage and accessed by the rules. For example, **person** would be a frame definition, and **mary** would be an instance of **person**.

The inheritance still works as before. That is, a **person** frame could be defined as well as **man** and **woman** frames, which inherit from **person**. In this case, **mary** would be an instance of **woman**, inheriting from both the **woman** frame and the **person** frame.

The **frame** definitions will be considered to define **classes** of things. So, **person**, **man**, and **woman** are classes defined in **frame** relations. Individuals, such as **mary**, **dennis**, **michael**, and **diana** are stored as instances of these classes.

To implement the instances, we need both a data structure and predicates to manipulate the data structure. An instance of a frame looks a lot like a frame, and will be stored in the relation **frinst/4**. The four arguments will be:

- the class name;
- the instance name;
- the list of slot attribute-value pairs associated with the instance;
- a time stamp indicating when the instance was last updated.

For example:

```
frinst(woman,
      mary,
      [ako-woman, hair-brown, hobby-rugby],
      32).
frinst(man,
      dennis,
      [ako-man, hair-blond, hobby-go],
      33).
```

The predicates which manipulate **frinsts** are:

- **getf** — retrieve attribute values for a **frinst**;
- **addf** — add a new **frinst**;
- **uptf** — update an existing **frinst**;
- **delf** — delete a **frinst**, or attribute values for a **frinst**;
- **printf** — print information about a **frinst**.

The code for **getf** is almost identical for that of **get_frame**. It just uses the **frinst** structure to get data rather than the **frame** structure. The **ako** slot of a **frinst** is automatically set to the **class** name, so if it is necessary to inherit values, the appropriate frames will be called just as they were for **get_frame**. The only other change is the additional argument for retrieving the time stamp as well.

```
getf(Class, Name, ReqList) :-
    getf(Class, Name, ReqList, _).
getf(Class, Name, ReqList, TimeStamp) :-
    frinst(Class, Name, SlotList, TimeStamp),
    slot_vals(Class, Name, ReqList, SlotList).
```

The **addf** predicate is similar to **add_frame**, however it has two new features. First, it will generate a unique name for the **frinst** if none was given, and second it adds a time stamp.

The generated name is simply a number in sequence. The time stamp is generated the same way, and uses the predicate **getchron**, which was already implemented for *Oops*. Note that **addf** also sets the **ako** slot to the value of the **Class**.

```
addf(Class, Nm, UList) :-
    (var(Nm), genid(Name);
     Name=Nm),
    add_slots(Class, Name, [ako-Class|UList], SlotList, NewList),
    getchron(TimeStamp),
    asserta(frinst(Class, Name, NewList, TimeStamp)),
    !.
```

The **up tf** predicate is distinct from **addf** in that it only updates existing **frinsts** and does not create new ones. It puts a new time stamp on the **frinst** as part of the update.

```
up tf(Class, Name, UList) :-
    frinst(Class, Name, SlotList, _),
    add_slots(Class, Name, UList, SlotList, NewList),
    retract( frinst(Class, Name, _, _) ),
    getchron(TimeStamp),
    asserta(frinst(Class, Name, NewList, TimeStamp)),
    !.
```

The **delf** and **printf** predicates are similarly based on **del_frame** and **print_frame**. Both offer options for accessing large numbers of instances. For example **delf(Class)** deletes all **frinsts** in **Class**, whereas **delf(Class, Name, DList)** deletes the attribute values in **DList** from the specified instance.

Rules for frinsts

Now that there is a mechanism for handling instances of frames, the next step is to revise the *Oops* rule structure to use those instances. In *Oops*, each of the **LHS** conditions was a Prolog term held in a **fact** relation. For *Foops*, the **LHS** conditions will be **frinsts**.

In keeping with the *Oops* design of using operators to make the rules more readable, the **frinsts** will be presented differently in the rules. The form will be:

```
Class - Name with [Attr - Val, ...]
```

For example, the rule in the furniture configuration that puts table lamps on end tables is:

```
rule fl1:
[table_lamp - TL with [position-none],
 end_table - ET with [position-wall/W]]
==>
[update( table_lamp - TL with [position-end_table/ET] )].
```

Note that the RHS actions also use the same syntax for the instance.

The change is easy to implement due to the interchangeability of facts and rules in Prolog. *Oops* refers to **facts**, expecting to find data. *Foops* uses the same code, but implements the relation **fact** as a rule, which calls **getf**.

Following is the code that matches the premises from the **LHS**. It is the the same as in the previous version except that the definition of **fact** has been changed to reflect the new nature of each individual premise.

```
match([], []).
```

```

match([Prem|Rest], [Prem/Time|InstRest]) :-
    mat(Prem, Time),
    match(Rest, InstRest).

mat(N:Prem, Time) :-
    !,
    fact(Prem, Time).
mat(Prem, Time) :-
    fact(Prem, Time).
mat(Test, 0) :-
    test(Test).

fact(Prem, Time) :-
    conv(Prem, Class, Name, ReqList),
    getf(Class, Name, ReqList, Time).

conv(Class-Name with List, Class, Name, List).
conv(Class-Name, Class, Name, []).

```

The **conv** relation is used to allow the user to specify instances in an abbreviated form if there is no attribute value list. For example, the following rule uses an instance of the class **goal** where the name is the only important bit of information:

```

rule f1:
[goal - couch_first,
 couch - C with [position-none, length-LenC],
 door - D with [position-wall/W],
 ...

```

The only other change which has to be made is in the implementation of the action commands which manipulate working storage. These now manipulate **frinst** structures instead of the pure facts as they did in *Oops*. They simply call the appropriate frame instance predicates.

```

assert_ws( fact(Prem, Time) ) :-
    conv(Prem, Class, Name, UList),
    addf(Class, Name, UList).

update_ws( fact(Prem, Time) ) :-
    conv(Prem, Class, Name, UList),
    uptf(Class, Name, UList).

retract_ws( fact(Prem, Time) ) :-
    conv(Prem, Class, Name, UList),
    delf(Class, Name, UList).

```

Adding Prolog to Foops

Now that frames and *Oops* have been integrated into a new system, *Foops*, the next step is to integrate Prolog as well. This has already been done for the frame based system with the various demons that can be associated with frame slots. The Prolog predicates referred to in the demon slots can simply be added directly to the knowledge base.

Adding Prolog to the rules is done by simply adding support for a call statement in both the **test** (for the LHS) and **take** (for the RHS) predicates.

```

...
test(call(X)) :- call(X).
...
...
take(call(X)) :- call(X).
...

```

Calls to Prolog predicates can now be added on either side of a rule. The Prolog can be simple predicates performing some function right in the knowledge base, or they can initiate more complex processing, including accessing other applications.

Figure 7.1 shows the major components of the *Foops* shell. Frames and Prolog code have been added to the knowledge base. Working storage is composed of frame instances, and the inference engine includes the frame manipulation predicates.

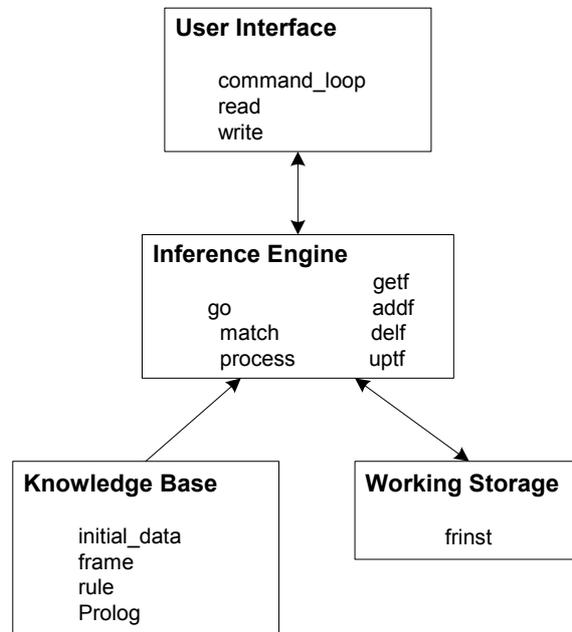


Figure 7.1 Major predicates of *Foops* shell

7.2 Room Configuration

Now that *Foops* is built, let's use it to attack the room configuration problem again. Many of the aspects of the original system were handled clumsily in the original version. In particular:

- The initial data gathering was awkward using rules that triggered other data gathering rules.
- The wall space calculations were done in the rules.
- Each rule was responsible for maintaining the consistency of the wall space and the furniture against the wall.

The new system will allow for a much cleaner expression of most of the system, and use Prolog to keep the rough edges out of the rule and frame knowledge structures.

Much of the knowledge about the furniture in the room is better stored in frames. This knowledge is then automatically applied by the rules accessing instances of furniture. The rules then become simpler, and just deal with the IF THEN situations and not data relationships.

Furniture frames

The knowledge base contains the basic frame definitions, which will be used by the instances of furniture. The frames act as sophisticated data definition statements for the system.

The highest frame defines the class furniture. It establishes defaults and demons that apply to all furniture.

```
frame(furniture,
      [legal_types - [val [couch,
                          chair,
                          coffee_table,
                          end_table,
                          standing_lamp,
                          table_lamp,
                          tv,
                          knickknack]],
      position - [def none, add pos_add],
      length - [def 3],
      place_on - [def floor]).
```

The most interesting slot is **position**. Each piece of furniture has the default of having the **position none**, meaning it has not been placed in the room. This means the programmer need not add this value for each piece of furniture initially. As it is positioned, the instance acquires a value that is used instead of the inherited default.

Also note that there is a demon, which is called when a **position** is added for a piece of furniture. This demon will automatically maintain the relation between wall space and furniture position. It will be described in detail a little later.

Next in the knowledge base are some classes of furniture. Note that the default length for a couch will override the general default length of 3 for any piece of furniture without a length specified.

```
frame(couch, [ako - [val furniture],
             length - [def 6]]).
frame(chair, [ako - [val furniture]]).
```

A table is another class of furniture, which is a bit different from other furniture in that things can be placed on a table. It has additional slots for available space, the list of items it is holding (things placed on the table), and the slot indicating that it can support other items.

```
frame(table, [ako - [val furniture],
             space - [def 4],
             length - [def 4],
             can_support - [def yes],
             holding - [def []]]).
```

There are two types of tables which are recognized in the system:

```
frame(end_table, [ako - [val table],
                 length - [def 2]]).
frame(coffee_table, [ako - [val table],
                   length - [def 4]]).
```

Remember that frames can have multiple inheritance paths. This feature can be used to establish other classes, which define attributes shared by certain types of furniture. For

example, the class **electric** is defined, which describes the properties of items that require electricity.

```
frame(electric, [needs_outlet - [def yes]]).
```

Lamps are electric items included in the knowledge base. Note that lamps are further divided between two types of lamps. A table lamp is different because it must be placed on a table.

```
frame(lamp, [ako - [val [furniture, electric]]]).
frame(standing_lamp, [ako - [val lamp]]).
frame(table_lamp, [ako - [val lamp],
                  place_on - [def table]]).
```

A knickknack is another item that should be placed on a table.

```
frame(knickknack, [ako - [val furniture],
                  length - [def 1],
                  place_on - [def table]]).
```

The television frame shows another use of calculated values. A television might be free standing or it might have to be placed on a table. This ambiguity is resolved by a calculate routine, which asks the user for a value. When a rule needs to know what to place the television on, the user will be asked. This creates the same kind of dialog effect seen in the backward chaining systems earlier in the book.

Note also that the television uses multiple inheritance, both as a piece of furniture and an electrical item.

```
frame(tv, [ako - [val [furniture, electric]],
          place_on - [calc tv_support]]).
```

Another frame defines walls. There is a slot for the number of outlets on the wall and the available space. If no space is defined, it is calculated. The holding slot is used to list the furniture placed against the wall.

```
frame(wall, [length - [def 10],
            outlets - [def 0],
            space - [calc space_calc],
            holding - [def []]]).
```

Doors, goals, and recommendations are other types of data that are used in the system.

```
frame(door, [ako - [val furniture],
            length - [def 4]]).
frame(goal, []).
frame(recommend, []).
```

Frame Demons

Next in the knowledge base are the Prolog predicates used in the various frame demons.

Here is the predicate that is called to calculate a value for the **place_on** slot for a television. It asks the user, and uses the answer to update the television **frinst** so that the user will not be asked again.

```
tv_support(tv, N, place_on-table) :-
    nl,
```

```

write('Should the TV go on a table? '),
read(yes),
uptf(tv, N, [place_on-table]).
tv_support(tv, N, place_on-floor) :-
uptf(tv, N, [place_on-floor]).

```

The **pos_add** demon is called whenever the **position** of a piece of furniture is updated. It illustrates how demons and frames can be used to create a database that maintains its own semantic integrity. In this case, whenever the position of a piece of furniture is changed, the available space of the wall it is placed next to, or the table it is placed on, is automatically updated. Also the holding list of the wall or table is also updated.

This means that the single update of a furniture position results in the simultaneous update of the wall or table space and wall or table holding list. Note the use of variables for the class and name make it possible to use the same predicate for both tables and walls.

```

pos_add(C, N, position-CP/P) :-
  getf(CP, P, [space-OldS]),
  getf(C, N, [length-L]),
  NewS is OldS - L,
  NewS >= 0,
  uptf(CP, P, [holding-[C/N], space-NewS]).
pos_add(C, N, position-CP/P) :-
  nl,
  write_line(['Not enough room on', CP, P, for, C, N]),
  !,
  fail.

```

This predicate also holds the pure arithmetic needed to maintain the available space. This used to be included in the bodies of the rules in *Oops*. Now it is only specified once, and is part of a demon defined in the highest frame, **furniture**. It never has to be worried about in any other frame definition or rules.

The **pos_add** demon also is designed to fail and report an error if something doesn't fit. The original **uptf** predicate, which was called to update the **position**, also fails, and no part of the update takes place. This insures the integrity of the database.

Initially, there is no space included in the wall and table **frinsts**. The following demon will calculate it based on the holding list. This could also have been used instead of the above predicate, but it is more efficient to calculate and store the number than to recalculate it each time.

```

space_calc(C, N, space-S) :-
  getf(C, N, [length-L, holding-HList]),
  sum_lengths(HList, 0, HLen),
  S is L - HLen.

sum_lengths([], L, L).
sum_lengths([C/N|T], X, L) :-
  getf(C, N, [length-HL]),
  XX is X + HL,
  sum_lengths(T, XX, L).

```

Initial Data

Now let's look at the data initialization for the system. It establishes other slots for the wall frames giving spatial relationships between them, and establishes the **goal gather_data**.

```

initial_data([goal - gather_data,
             wall - north with [opposite-south,
                               right-west,

```

```

        left-east],
    wall - south with [opposite-north,
        right-east,
        left-west],
    wall - east with [opposite-west,
        right-north,
        left-south],
    wall - west with [opposite-east,
        right-south,
        left-north]
]).

```

Input Data

The first rule uses the call feature to call a Prolog predicate to perform the data gathering operations, which used to be done with rules in *Oops*. *Foops* uses the *Lex* rule selection, but this rule will fire first because no other rules have any furniture to work with. It then asserts the **goal couch_first** after gathering the other data. Because *Lex* gives priority to rules accessing recently updated elements in working storage, the rules that have as a **goal couch_first** will fire next.

```

rule 1:
[goal - gather_data]
==>
[call(gather_data),
 assert( goal - couch_first )].

```

The Prolog predicate proceeds with prompts to the user and calls to frame predicates to populate working storage.

```

gather_data :-
    read_furniture,
    read_walls.

read_furniture :-
    get_frame(furniture, [legal_types-LT]),
    write('Enter name of furniture at the prompt. '), nl,
    write(' It must be one of:'), nl,
    write(LT), nl,
    write('Enter end to stop input. '), nl,
    write('At the length prompt enter y or a new number. '), nl,
    repeat,
        write('>'),
        read(X),
        process_furn(X),
    !.

```

Note that this predicate has the additional intelligence of finding the default value for the length of a piece of furniture and allowing the user to accept the default, or choose a new value.

```

process_furn(end).
process_furn(X) :-
    get_frame(X, [length-DL]),
    write(length-DL), write('>'),
    read(NL),
    get_length(NL, DL, L),
    addf(X, _, [length-L]),
    fail.

get_length(y, L, L) :- !.
get_length(L, _, L).

```

The dialog to get the empty room layout is straight-forward Prolog.

```
read_walls :-
    nl, write('Enter data for the walls. '), nl,
    write('What is the length of the north & south walls? '),
    read(NSL),
    uptf(wall, north, [length-NSL]),
    uptf(wall, south, [length-NSL]),
    write('What is the length of the east & west walls? '),
    read(EWL),
    uptf(wall, east, [length-EWL]),
    uptf(wall, west, [length-EWL]),
    write('Which wall has the door? '),
    read(DoorWall),
    write('What is its length? '),
    read(DoorLength),
    addf(door, D, [length-DoorLength]),
    uptf(door, D, [position-wall/DoorWall]),
    write('Which walls have outlets? (a list)'),
    read(PlugWalls),
    process_plugs(PlugWalls).

process_plugs([]) :- !.
process_plugs([H|T]) :-
    uptf(wall, H, [outlets-1]),
    !,
    process_plugs(T).
process_plugs(X) :-
    uptf(wall, X, [outlets-1]).
```

The Rules

With the data definition, initial data, and input taken care of, we can proceed to the body of rules. They are much simpler than the original versions.

The first rules place the couch either opposite the door or to its right, depending on which wall has more space. Note that the update of the couch position is done with a single action. The frame demons take care of the rest of the update.

```
rule f1:
[goal - couch_first,
 couch - C with [position-none, length-LenC],
 door - D with [position-wall/W],
 wall - W with [opposite-OW, right-RW],
 wall - OW with [space-SpOW],
 wall - RW with [space-SpRW],
 SpOW >= SpRW,
 LenC =< SpOW]
==>
[update(couch - C with [position-wall/OW])].

rule f2:
[goal - couch_first,
 couch - C with [position-none, length-LenC],
 door - D with [position-wall/W],
 wall - W with [opposite-OW, right-RW],
 wall - OW with [space-SpOW],
 wall - RW with [space-SpRW],
 SpRW >= SpOW,
 LenC =< SpRW]
==>
[update(couch - C with [position-wall/RW])].
```

The next rules position the television opposite the couch. They cover the two cases of a free standing television and one that must be placed on a table. If the television needs to be placed on a table, and there is no table big enough, then a recommendation to buy an end table for the television is added. Because of specificity in *Lex* (the more specific rule has priority), rule f4 will fire before f4a. If f4 was successful, then f4a will no longer apply. If f4 failed, then f4a will fire the next time.

The rule to position the television puts the end table on the wall opposite the couch and the television on the end table.

```
rule f3:
[couch - C with [position-wall/W],
 wall - W with [opposite-OW],
 tv - TV with [position-none, place_on-floor]]
==>
[update(tv - TV with [position-wall/OW])].

rule f4:
[couch - C with [position-wall/W],
 wall - W with [opposite-OW],
 tv - TV with [position-none, place_on-table],
 end_table - T with [position-none]]
==>
[update(end_table - T with [position-wall/OW]),
 update(tv - TV with [position-end_table/T])].

rule f4a:
[tv - TV with [position-none, place_on-table]]
==>
[assert(recommend - R with [buy-['table for tv']])].
```

The coffee table should be placed in front of the couch, no matter where it is.

```
rule f5:
[coffee_table - CT with [position-none], couch - C]
==>
[update(coffee_table - CT with [position-frontof(couch/C)])].
```

The chairs go on adjacent walls to the couch.

```
rule f6:
[chair - Ch with [position-none],
 couch - C with [position-wall/W],
 wall - W with [right-RW, left-LW],
 wall - RW with [space-SpR],
 wall - LW with [space-SpL],
 SpR > SpL]
==>
[update(chair - Ch with [position-wall/RW])].

rule f7:
[chair - Ch with [position-none],
 couch - C with [position-wall/W],
 wall - W with [right-RW, left-LW],
 wall - RW with [space-SpR],
 wall - LW with [space-SpL],
 SpL > SpR]
==>
[update(chair - Ch with [position-wall/LW])].
```

The end tables go next to the couch if there are no other end tables there. Otherwise they go next to the chairs. Note that the rule first checks to make sure there isn't an unplaced television that needs an end table for support. The television rule will position the end table for holding the television.

```

rule f9:
[end_table - ET with [position-none],
 not tv - TV with [position-none, place_on-table],
 couch - C with [position-wall/W],
 not end_table - ET2 with [position-wall/W]]
==>
[update(end_table - ET with [position-wall/W])].

rule f10:
[end_table - ET with [position-none],
 not tv - TV with [position-none, place_on-table],
 chair - C with [position-wall/W],
 not end_table - ET2 with [position-wall/W]]
==>
[update(end_table - ET with [position-wall/W])].

```

Table lamps go on end tables.

```

rule f11:
[table_lamp - TL with [position-none],
 end_table - ET with [position-wall/W]]
==>
[update( table_lamp - TL with [position-end_table/ET] )].

```

Knickknacks go on anything that will hold them. Note the use of variables in the class and name positions. The query to the slot **can_support** will cause this rule to find anything which has the attribute value **can_support - yes**. This slot is set in the table frame, so both end tables and coffee tables will be available to hold the knickknack.

```

rule f11a:
[knickknack - KK with [position-none],
 Table - T with [can_support=yes, position-wall/W]]
==>
[update( knickknack - KK with [position-Table/T] )].

```

The rules for determining if extensions cords are necessary are simplified by the use of variables and frame inheritance. The rule looks for anything that needs an outlet. This will be true of any items that need an outlet, which is a property inherited from frame **electric**. It is not necessary to write separate rules for each case.

It is necessary to write a separate rule to cover those things that are positioned on other things. The wall can only be found from the supporting item. This is the case where a television or table lamp is placed on a table. While this is handled in rules here, it would also have been possible to use frame demons to cover this case instead.

```

rule f12:
[Thing - X with [needs_outlet=yes, position-wall/W],
 wall - W with [outlets-0]]
==>
[assert(recommend - R with [buy-['extension cord'-W]])].

rule f13:
[Thing - X with [needs_outlet=yes, position-C/N],
 C - N with [position-wall/W],
 wall - W with [outlets-0]]
==>
[assert(recommend - R with [buy-['extension cord'-Thing/W]])].

```

Due to specificity priorities in *Lex*, the following rule will fire last. It calls a Prolog predicate to output the results of the session.

```

rule f14:
[]

```

```
==>
[call(output_data)].
```

Output Data

The **output_data** predicate is again straight forward Prolog, which gets the relevant information and displays it.

```
output_data :-
    write('The final results are:'), nl,
    output_walls,
    output_tables,
    output_recommends,
    output_unplaced.

output_walls :-
    getf(wall, W, [holding-HL]),
    write_line([W, wall, holding|HL]),
    fail.
output_walls.

output_tables :-
    getf(C, N, [holding-HL]),
    not C = wall,
    write_line([C, N, holding|HL]),
    fail.
output_tables.

output_recommends :-
    getf(recommend, _, [buy-BL]),
    write_line([purchase|BL]),
    fail.
output_recommends.

output_unplaced :-
    write('Unplaced furniture:'), nl,
    getf(T, N, [position-none]),
    write(T-N), nl,
    fail.
output_unplaced.
```

Figure 7.2 summarizes how the tools in *Foops* are applied to the furniture layout program. Frames are used for objects and relationships, rules are used to define situations and responses, and Prolog is used for odds and ends like I/O and calculations.

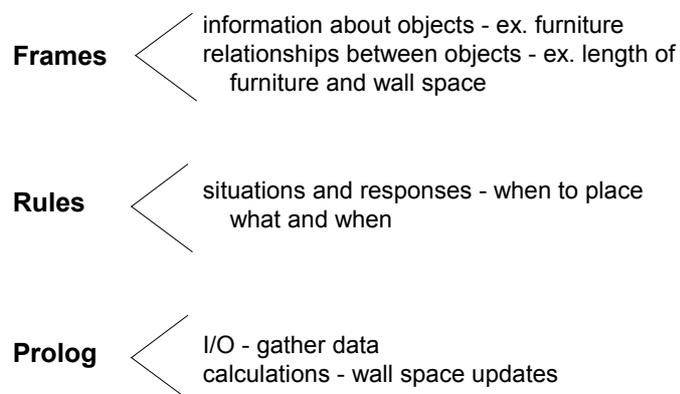


Figure 7.2 Summary of knowledge representation tools used in *Foops*

7.3 A Sample Run

Here is a portion of a sample run of the furniture placement system.

The system starts in the *Foops* command loop, and then begins the initial data gathering.

```
=>go.
```

```
Enter name of furniture at the prompt. It must be one of:
[couch, chair, coffee_table, end_table, standing_lamp, table_lamp,
tv, knickknack]
```

```
Enter end to stop input.
```

```
At the length prompt enter y or a new number.
```

```
>couch.
```

```
length-6>y.
```

```
>chair.
```

```
length-3>5.
```

```
...
```

```
>end.
```

```
Enter data for the walls.
```

```
What is the length of the north & south walls? 12.
```

```
What is the length of the east & west walls? 9.
```

```
Which wall has the door? east.
```

```
What is its length? 3.
```

```
Which walls have outlets? (a list)[east].
```

```
adding-(goal-couch_first)
```

```
Rule fired 1
```

One of the rules accessing the television causes this prompt to appear.

```
Should the TV go on a table? yes.
```

The system has informational messages regarding which rules are firing and what data is being updated.

```
updating-(couch-110 with [position-wall/north])
```

```
Rule fired f2
```

```
updating-(end_table-116 with [position-wall/south])
```

```
updating-(tv-117 with [position-end_table/116])
```

```
Rule fired f4
```

```
...
```

Here is a message that appeared when a knickknack was unsuccessfully placed on an end table. A different knickknack was then found to fit on the same table.

```
Not enough room on end_table 116 for knickknack 121
```

```
Rule fired f11a
```

```
updating-(knickknack-120 with [position-end_table/116])
```

```
Rule fired f11a
```

Here is one of the extension cord recommendations:

```
adding-(recommend-_3888 with [buy-[extension_cord-
                                table_lamp/north]])
```

```
Rule fired f13
```

The last rule to fire provides the final results.

```
The final results are:
north wall holding end_table/114 couch/110
east wall holding chair/112 door/122
west wall holding end_table/115 chair/113
south wall holding end_table/116
end_table 114 holding table_lamp/119
end_table 115 holding knickknack/121
end_table 116 holding knickknack/120 tv/117
purchase extension cord-table_lamp/north
purchase extension cord-tv/south
Unplaced furniture:
table_lamp-118
chair-111
```

7.4 Summary

A combination of techniques can lead to a much cleaner representation of knowledge for a particular problem. The Prolog code for each of the techniques can be integrated relatively easily to provide a more complex system.

Exercises

- 7.1 Integrate *Clam* with frames.
- 7.2 Implement multiple rule sets as described in the chapter five exercises. Let each rule set be either forward or backward chaining, and use the appropriate inference engine for both.
- 7.3 Build another expert system using *Foops*.

8 Performance

As the size of a knowledge base grows, performance becomes more problematic. The inference engines we have looked at so far use a simple pattern matching algorithm to find the rules to fire. Various indexing schemes can be used to speed up that pattern matching process.

The indexing problem is different for forward and backward chaining systems. Backward chaining systems need to be accessed by the goal pattern in the right hand side of the rule. Forward chaining systems need to be indexed by the more complex patterns on the left hand side. Backward chaining issues will be discussed briefly in this chapter, followed by more in-depth treatment of a simplified Rete match algorithm for the *Foops* forward chaining system.

8.1 Backward Chaining Indexes

For performance in backward chaining systems, the rules are indexed by goal patterns on the right hand side. In particular, if the goal is to find a value for a given attribute, then the rules should be indexed by the attribute set in the RHS of the rule. This happens automatically for the pure Prolog rules in the bird identification program since the attributes are Prolog predicate names, which are generally accessed through hashing algorithms. The indices, if desired, must be built into the backward chaining engine used in *Clam*. Some Prologs provide automatic indexing on the first argument of a predicate. This feature could be used to provide the required performance.

Given indexing by the first argument, the rules in *Clam* would be represented by:

```
rule(Attribute, Val, CF, Name, LHS).
```

This way, a search for rules providing values for a given attribute would quickly find the appropriate rules.

Without this, each rule could be represented with a functor based on the goal pattern and accessed using the univ (=..) predicate rather than the pattern matching currently used in *Clam*. The predicates that initially read the rules can store them using this scheme. For example, the internal format of the *Clam* rules would be:

```
attribute(Val, CF, Name, LHS).
```

In particular, some rules from the car diagnostic system would be:

```
problem(battery, 100, 'rule 1',
        [av(turn_over, no), av(battery_bad, yes)]).
problem(flooded, 80, 'rule 4',
        [av(turn_over, yes), av(smell_gas, yes)]).

battery_bad(yes, 50, 'rule 3', [av(radio_weak, yes)]).
```

When the inference is looking for rules to establish values for an attribute-value pattern, **av(A, V)**, the following code would be used:

```
Rule =.. [A, V, CF, ID, LHS],
call(Rule),
...
```

This structure would allow *Clam* to take advantage of the hashing algorithms built into Prolog for accessing predicates.

8.2 Rete Match Algorithm

OPS5 and some high-end expert system shells use the Rete match algorithm to optimize performance. It is an indexing scheme, which allows for rapid matching of changes in working memory with the rules. Previous match information is saved on each cycle, so the system avoids redundant calculations. We will implement a simplified version of the Rete algorithm for *Foops*.

The Rete algorithm is designed for matching left hand side rule patterns against working storage elements. Let's look at two similar rules from the room placement expert system.

```
rule f3#
[couch - C with [position-wall/W],
 wall - W with [opposite-OW],
 tv - TV with [position-none, place_on-floor]]
==>
[update(tv - TV with [position-wall/OW])].

rule f4#
[couch - C with [position-wall/W],
 wall - W with [opposite-OW],
 tv - TV with [position-none, place_on-table],
 end_table - T with [position-none]]
==>
[update(end_table - T with [position-wall/OW]),
 update(tv - TV with [position-end_table/T])].
```

First let's define some terms. Each LHS is composed of one or more frame patterns. An example of a frame pattern is:

```
tv-TV with [position-none, place_on-table]
```

The frame pattern will be the basic unit of indexing in the simplified Rete match algorithm. In a full implementation, indexing is carried down to the individual attribute-value pairs within the frame pattern, such as **place_on-table**.

The match algorithm used in the first implementation of *Foops* takes every rule and compares it to all the frame instances on each cycle. In particular, both of the example rules above would be compared against working storage on each cycle. Not only is redundant matching being done on each cycle, within each cycle the same frame patterns are being matched twice, once for each rule. Since working memory generally has few changes on each cycle, and since many of the rules have redundant patterns, this approach is very inefficient.

With the Rete algorithm, the status of the match information from one cycle is remembered for the next. The indexing allows the algorithm to update only that match information which is changed due to working memory changes.

The rules are compiled into a network structure where the nodes correspond to the frame patterns in the LHS of the rules. There are three basic types of node, which serve as: the entry to the network; the internals of the network; and the exit from the network. These are called root nodes, two-input nodes, and rule nodes respectively.

The network has links which run from single frame patterns in the root nodes, through the two-input nodes, to full LHS patterns in the rule nodes. Figure 8.1 shows the nodes and links generated from the two sample rules.

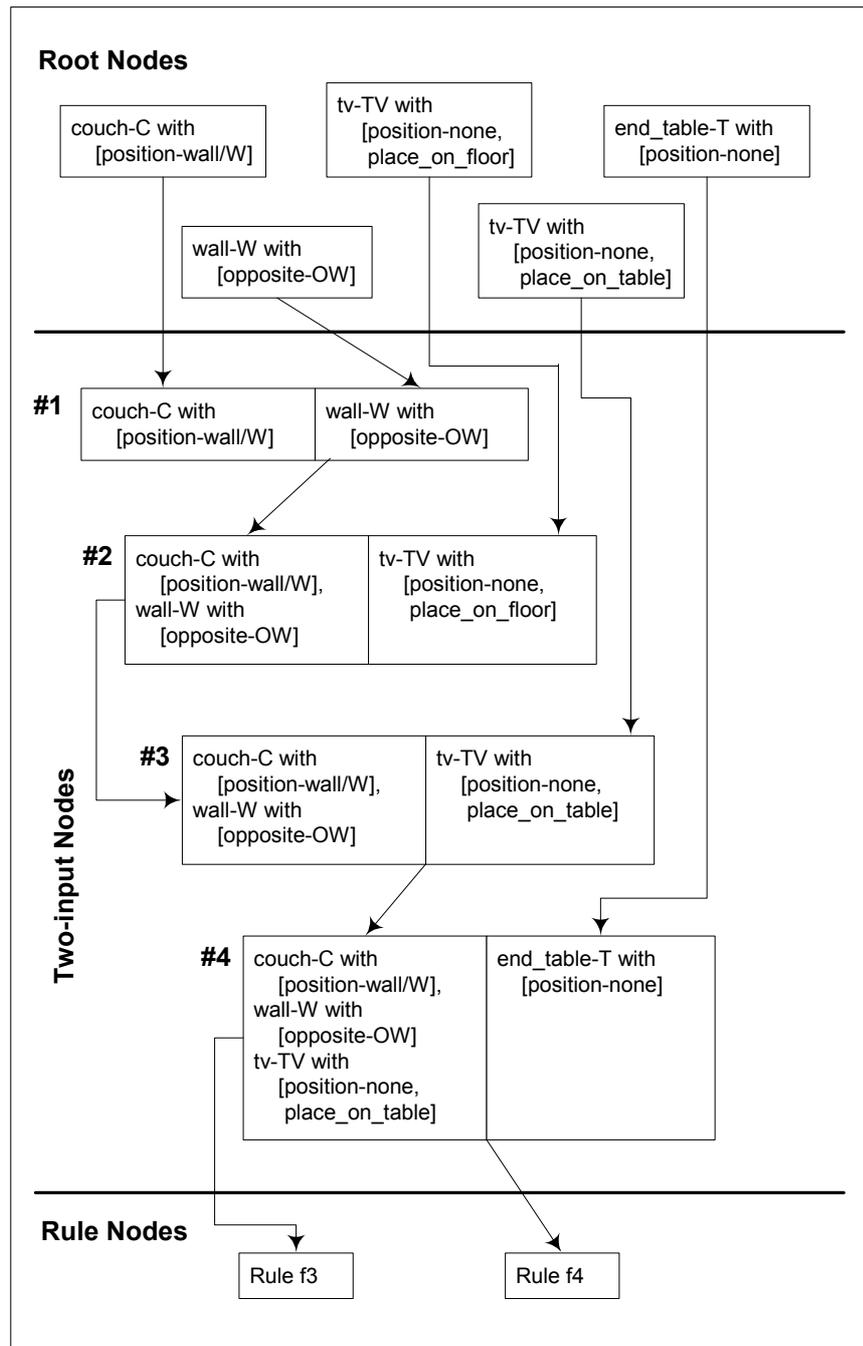


Figure 8.1 The nodes of the Rete network for two sample rules

Network Nodes

The root nodes serve as entry points to the Rete network. They are the simplest patterns recognized by the network, in this case the frame patterns that appear in the various rules. A frame pattern only appears once in a root node even if it is referenced in multiple rules. Each root node has pointers to two-input nodes, which are used to combine the patterns into full LHS patterns.

Two-input nodes represent partially completed LHS patterns. The left input has a pattern that has one or more frame patterns as they appear in one or more rules. The right input has a single frame pattern, which when appended to the left input pattern completes more

of a full LHS pattern. The two-input node is linked to other two-input or rule nodes whose left input matches the larger pattern.

The rule nodes are the exit points of the Rete network. They have full LHS patterns and RHS patterns.

Network Propagation

Associated with each two-input node are copies of working storage elements that have already matched either side of the node. These are called the left and right memories of the node. In effect, this means working memory is stored in the network itself.

Whenever a frame instance is added, deleted, or updated it is converted to a "token". A token is formed by comparing the frame instance to the root patterns. A root pattern which is unified with the frame instance is a token. The token has an additional element which indicates whether it is to be added to or deleted from the network.

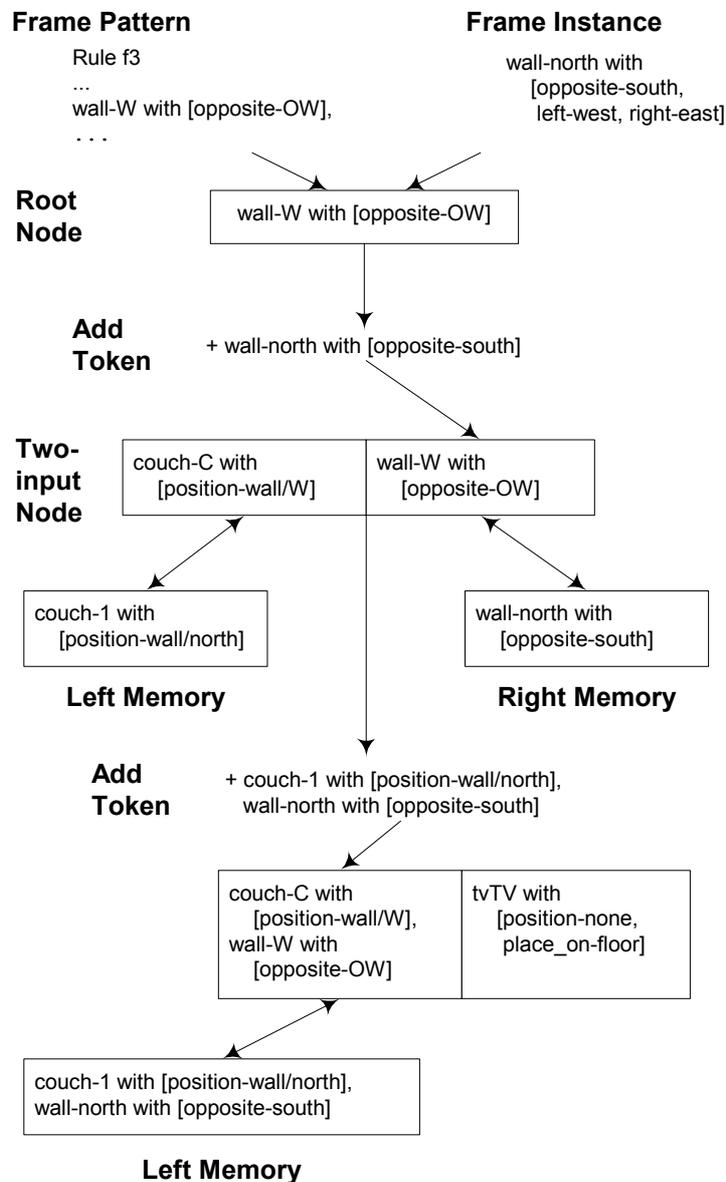


Figure 8.2 The relationship between frame patterns, instances, tokens, and nodes

The token is sent from the root node to its successor nodes. If the token matches a left or right pattern of a two-input successor node, it is added (or deleted) from the the appropriate memory for that node. The token is then combined with each token from the memory on the other side (right or left) and compared with the pattern formed by combining the left and right patterns of the two input node. If there is a match, the new combined token is sent to the successor nodes. This process continues until either a rule is instantiated or there are no more matches.

Figure 8.2 shows the relationships between rules, frame patterns, frame instances, nodes, and tokens. It shows the top portion of the network as shown in Figure 8.1. It assumes that couch-1 with [position-wall/north] already exists in the left memory of two-input node #1. Then the frame instance wall-north with [opposite-south, left-west, right-east] is added, causing the generation of the token wall-north with [opposite-south]. The token updates the right memory of node #1 as shown, and causes a new token to be generated, which is sent to node #2, causing an update to its left memory.

Example of Network Propagation

Lets trace what happens in the network during selected phases of a run of the room configuration system.

First the walls are placed during initialization. There are four wall frame instances asserted, which define opposites and are therefore recognized by the portion of the system we are looking at. They are used to build add-tokens that are sent to the network.

```
wall-north with [opposite-south].
wall-south with [opposite-north].
wall-east with [opposite-west].
wall-west with [opposite-east].
```

Each of these tokens matches the following root pattern, binding **OW** to the various directions:

```
wall-W with [opposite-OW].
```

Therefore, each token is passed on to the right side of two-input node #1 as indicated by the list of links associated with that root pattern. Each of these terms is stored in the right memory of node #1. Since there is nothing in the left memory of node #1, network processing stops until the next input is received.

Next, the furniture is initialized, with the couch, tv, and end_table placed with **position-none**. They will be internally numbered 1, 2, and 3. Since the root pattern for couch in the segment we are looking at has a **position-wall/W**, the couch does not show up in it at this time. However, node #2 and node #4 have their right memories updated respectively with the tokens:

```
tv-2 with [position-none, place_on-floor].
end_table-3 with [position-none].
```

At this point the system looks like figure 8.3. The shaded boxes correspond to the two-input nodes of figure 8.1.

Two-input Nodes

Left Memory

Right Memory

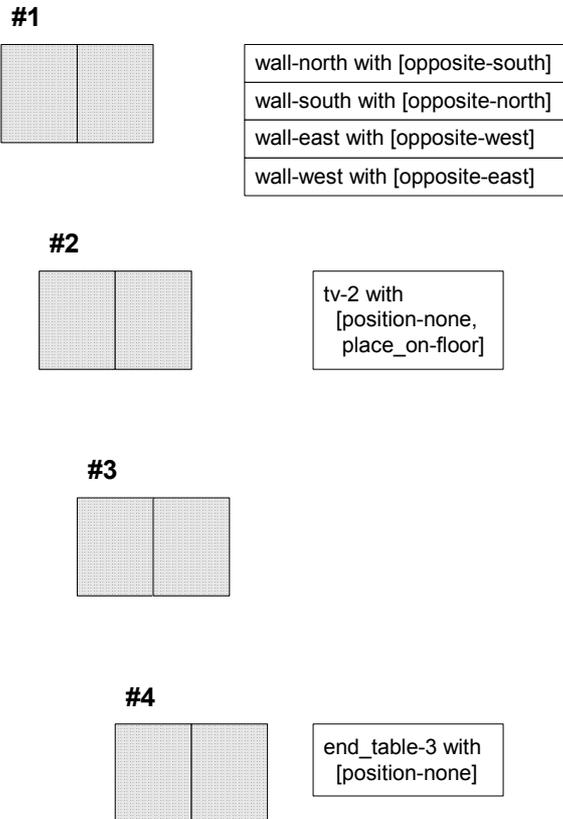


Figure 8.3 The sample network after initialization

After initialization, the system starts to fire rules. One of the early rules in the system will lead to the placing of the couch against a wall, for example the north wall. This update will cause the removal of the token **couch-1 with [position-none]** from parts of the network not shown in the diagrams, and the addition of the token **couch-1 with [position-wall/north]** to the left memory of node #1 as shown in figure 8.4. This causes a cascading update of various left memories as shown in the rest of figure 8.4 and described below.

Node #1 now has both left and right memories filled in, so the system tries to combine the one new memory term with the four right memory terms. There is one successful combination with the **wall-north** token, so a new token is built from the two and passed to the two successor nodes of node #1. The new token is:

```
[couch-1 with [position-wall/north],
 wall-north with [opposite-south] ]
```

This token is stored in the left memories of both successors, node #2 and node #3. There is no right memory in node #3, so nothing more happens there, but there is right memory in node #2 that does match the input token. Therefore a new token is constructed and sent to the successor of node #2. The new token is:

```
[couch-1 with [position-wall/north],
 wall-north with [opposite-south],
 tv-2 with [position-none, place_on-floor] ]
```

The token is sent to the successor, which is rule #f3. The token is the binding of the left side of the rule, and leads to variable bindings on the right side of the rule. This is a full

instantiation of the rule and is added to the conflict set. When the rule is fired, the action on the right side causes the position of the tv to be updated.

```
update ( tv-2 with [position-wall/south] )
```

Two-input Nodes

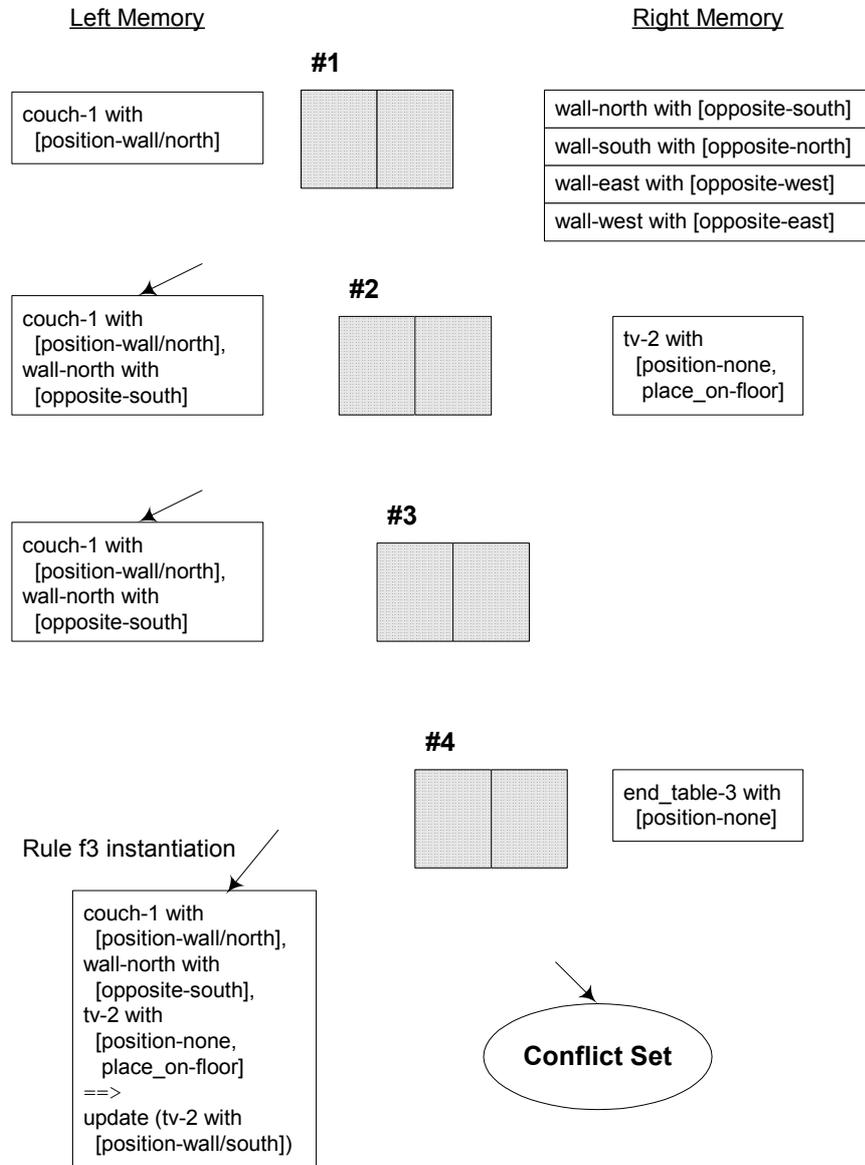


Figure 8.4 The cascading effect of positioning the couch

This update causes two tokens to be sent through the network. One is a delete token for **tv-2 with [position-none]**, and the other is an add token for **tv-2 with [position-wall/south]**. The delete causes the removal of the token from the right memory of node#2. The add would not match any patterns in this segment of the network.

Performance Improvements

The Rete network provides a number of performance improvements over a simple matching of rules to working memory:

- The root nodes act as indices into the network. Only those nodes affected by an update to working memory are processed.
- The patterns that have been successfully matched are saved in the node left and right memories. They do not have to be reprocessed.
- When rules share common sequences of patterns, that information is stored only once in the network, meaning it only has to be processed once.
- The output of the network is full rule instantiations. Once an instantiation is removed from the conflict set (due to a rule firing) it will not reappear on the conflict set — thus preventing multiple firings of the rule.

Next, let's look at the code to build a Rete pattern matcher. First we will look at the data structures used to define the Rete network, then the pattern matcher which propagates tokens through the network, and finally the rule compiler that builds the network from the rules.

8.3 The Rete Graph Data Structures

The roots of the network are based on the frame patterns from the rules. The root nodes are represented as:

```
root(NID, Pattern, NextList).
```

NID is a generated identification for the node, **Pattern** is the frame pattern, and **NextList** is the list of successor nodes that use the **Pattern**. **NextList** serves as the pointers connecting the network together. For example:

```
root(2, wall-W with [opposite-OW], [1-r]).
```

The two-input nodes of the network have terms representing the patterns that are matched from the left and right inputs to the node. Together they form the template, which determines if particular tokens will be successfully combined into rule instantiations. The format of this structure is:

```
bi(NID, LeftPattern, RightPattern, NextList).
```

NID again is an identification. **LeftPattern** is the list of frame patterns that have been matched in nodes previous to this one. **RightPattern** is the new frame pattern that will be appended to the **LeftPattern**. **NextList** contains a list of successor nodes. For example:

```
bi(1, [couch-C with [position-wall/W]],
     [wall-W with [opposite-OW],
      [2-1, 3-1]]).
bi(2, [couch-C with [position-wall/W],
     wall-W with [opposite-OW]],
     [tv-TV with [position-none, place_on-floor]],
     [rule(f3)]).
```

The end of the network is rules. The rules are stored as:

```
rul(N, LHS, RHS).
```

N is the identification of the rule. **LHS** is the list of frame patterns which represent the full left hand side of the rule. **RHS** is the actions to be taken when the rule is instantiated.

8.4 Propagating Tokens

Tokens are generated from the updates to frame instances. There are only two update predicates for the network: **addrete**, which adds tokens, and **delrete**, which deletes them. Both take as input the **Class**, **Name**, and **TimeStamp** of the frame instance. Both are called from the *Foops* predicates that update working memory: **assert_ws**, **retract_ws** and **update_ws**. The major predicates of **addrete** are shown in figure 8.5.

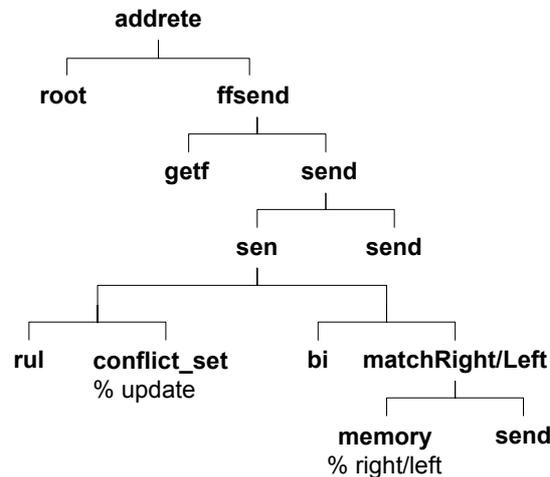


Figure 8.5 Major predicates which propagate a token through the network

The **addrete** predicate uses a simple repeat-fail loop to match the frame instance against each of the root nodes. It looks like:

```

addrete(Class, Name, TimeStamp) :-
    root(ID, Class-Name with ReqList, NextList),
    ffsend(Class, Name, ReqList, TimeStamp, NextList),
    fail.
addrete(_, _, _).
  
```

The **ffsend** predicate fulfills the request pattern in the **root** by a call to the frame system predicate, **getf**. This fills in the values for the pattern thus creating a token. Next, **send** is called with an add token.

```

ffsend(Class, Name, ReqList, TimeStamp, NextList) :-
    getf(Class, Name, ReqList),
    send(tok(add, [(Class-Name with ReqList)/TimeStamp]), NextList),
    !.
  
```

The **delrete** predicate is analagous, the only difference being it **sends** a delete token through the network.

```

delrete(Class, Name, TimeStamp) :-
    root(ID, Class-Name with ReqList, NextList),
    delr(Class, Name, ReqList, TimeStamp),
    fail.
delrete(_, _, _).

delr(Class, Name, ReqList, TimeStamp) :-
    getf(Class, Name, ReqList),
    !,
    send(tok(del, [(Class-Name with ReqList)/TimeStamp]), NextList).
delr(Class, Name, ReqList, TimeStamp).
  
```

Predicate **send** passes the token to each of the successor nodes in the list:

```
send(_, []).
send(Token, [Node|Rest]) :-
    sen(Node, Token),
    send(Token, Rest).
```

The real work is done by **sen**. It has to recognize three cases:

- The token is being sent to a rule. In this case, the rule must be added to or deleted from the conflict set.
- The token is being sent to the left side of a two-input node. In this case, the token is added to or deleted from the left memory of the node. The list is then matched against the right memory elements to see if a larger token can be built and passed through the network.
- The token is being sent to the right side of a node. In this case, the token is added to or deleted from the right memory of the node. It is then compared against the left memory elements to see if a larger token can be built and passed through the network.

In Prolog:

```
sen(rule-N, tok(AD, Token)) :-
    rul(N, Token, Actions),
    (AD = add, add_conflict_set(N, Token, Actions);
     AD = del, del_conflict_set(N, Token, Actions)),
    !.
sen(Node-l, tok(AD, Token)) :-
    bi(Node, Token, Right, NextList),
    (AD = add, asserta(memory(Node-l, Token));
     AD = del, retract(memory(Node-l, Token))),
    !,
    matchRight(Node, AD, Token, Right, NextList).
sen(Node-r, tok(AD, Token)) :-
    bi(Node, Left, Token, NextList),
    (AD = add, asserta(memory(Node-r, Token));
     AD = del, retract(memory(Node-r, Token))),
    !,
    matchLeft(Node, AD, Token, Left, NextList).
```

Note how Prolog's unification automatically takes care of variable bindings between the patterns in the node memory, and the instance in the token. In **sen**, the instance in **Token** is unified with one of the right or left patterns in **bi**, automatically causing the opposite pattern to become instantiated as well (or else the call to **bi** fails and the next **bi** is tried). This instantiated **Right** or **Left** is then sent to **matchRight** or **matchLeft** respectively.

Both **matchRight** and **matchLeft** take the instantiated **Right** or **Left** and compare it with the tokens stored in the right or left working memory associated with that node. If unification is successful, a new token is built by appending the right or left from the memory with the original token. The new token is then passed further with another call to **send**.

```
matchRight(Node, AD, Token, Right, NextList) :-
    memory(Node-r, Right),
    append(Token, Right, NewTok),
    send(tok(AD, NewTok), NextList),
    fail.
matchRight(_, _, _, _, _).

matchLeft(Node, AD, Token, Left, NextList) :-
    memory(Node-l, Left),
    append(Left, Token, NewTok),
```

```

    send(tok(AD, NewTok), NextList),
    fail.
matchLeft(_, _, _, _, _).

```

Another type of node that is useful for the system handles the cases where the condition on the LHS of the rule is a test, such as $L > R$ or **member(X, Y)** – rather than a pattern to be matched against working memory. The test nodes just perform the test and pass the tokens through if they succeed. There is no memory associated with them.

A final node to make the system more useful is one to handle negated patterns in rules. It works as a normal node, keeping instances in its memory that match the pattern in the rule, however it only passes on tokens which do not match.

8.5 The Rule Compiler

The rule compiler builds the Rete network from the rules. The compiler predicates are not as straight forward as the predicates that propagate tokens through the network. This is one of the rare cases where the power of Prolog's pattern matching actually gets in the way, and code needs to be written to overcome it.

The very unification which makes the pattern matching propagation predicates easy to code gets in the way of the rule compiler. We allow variables in the rules, which behave as normal Prolog variables, but when the network is built, we need to know which rules are matching variables and which are matching atoms. For example, one rule might have the pattern **wall/W** and another might have **wall/east**. These should generate two different indices when building the network, but Prolog would not distinguish between them since they unify with each other.

In order to distinguish between the variables and atoms in the frame patterns, we must pick the pattern apart first, binding the variables to special atoms as we go. Once all of the variables have been instantiated in this fashion, the patterns can be compared.

But first, let's look at the bigger picture. Each rule is compared, frame pattern by frame pattern, against the network which has been developed from the rules previously processed. The frame patterns of the rule are sent through the network in a similar fashion to the propagation of tokens. If the frame patterns in the rule are accounted for in the network, nothing is done. If a pattern is not in the network, then the network is updated to include it.

The top level predicate for compiling rules into a Rete net, reads each rule and compiles it.

```

rete_compil :-
    rule N# LHS ==> RHS,
    rete_comp(N, LHS, RHS),
    fail.
rete_compil.

```

The major predicates involved in compiling rules into a Rete network are shown in figure 8.6.

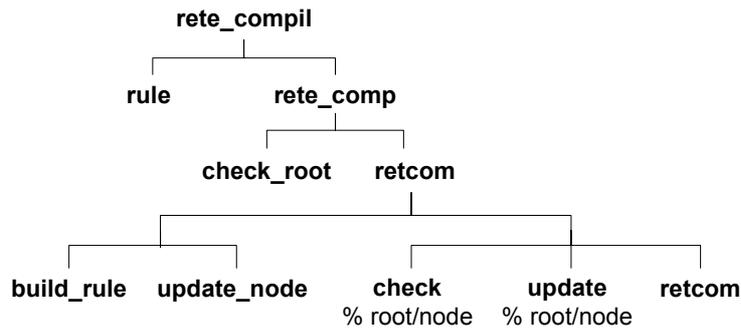


Figure 8.6 The major predicates for compiling rules into a Rete network

The next predicate, **rete_comp**, looks at the first frame pattern in the rule and determines if it should build a new root node or if one already exists. It then passes the information from the root node and the rest of the rule left hand side to **retcom**, which continues traversing and/or building the network.

```

rete_comp(N, [H|T], RHS) :-
    term(H, Hw),
    check_root(RN, Hw, HList),
    retcom(root(RN), [Hw/_], HList, T, N, RHS),
    !.
rete_comp(N, _, _) .
  
```

The **term** predicate merely checks for shorthand terms of the form **Class-Name** and replaces them with the full form **Class-Name with []**. Terms already in full form are left unchanged.

```

term(Class-Name, Class-Name with []).
term(Class-Name with List, Class-Name with List).
  
```

The **check_root** predicate determines if there is already a root node for the term and, if not, creates one. It will be described in detail a little later. The third argument from **check_root** is the current list of nodes linked to this root.

The last goal is to call **retcom**, which is the main recursive predicate of the compilation process. It has six arguments, as follows:

- PNID** — the id of the previous node
- OutPat** — pattern from previous node
- PrevList** — successor list from previous node
- [H|T]** — list of remaining frame patterns in rule
- N** — rule ID, for building the rule at the end
- RHS** — RHS of the rule, for building the rule at the end

There are two cases recognized by **retcom**:

- All of the frame patterns in the rule have been compiled into the network, and all that is left is to link the full form of the rule to the network.
- The rule frame patterns processed so far match an existing two-input node, or a new one is created.

In Prolog, the first case is recognized by having the empty list as the list of remaining frame patterns. The rule is built and **update_node** is called to link the previous node to the rule.

```
retcom(PNID, OutPat, PrevList, [], N, RHS) :-
    build_rule(OutPat, PrevList, N, RHS),
    update_node(PNID, PrevList, rule-N),
    !.
```

In the second case, the frame pattern being worked on (**H**) is first checked to see if it has a root node. Then the network is checked to see if a two-input node exists whose left input pattern matches the rule patterns processed so far (**PrevNode**). Either node might have been found or added, and then linked to the rest of the network.

```
retcom(PNID, PrevNode, PrevList, [H|T], N, RHS) :-
    term(H, Hw),
    check_root(RN, Hw, HList),
    check_node(PrevNode, PrevList, [Hw/_], HList, NID,
              OutPat, NList),
    update_node(PNID, PrevList, NID-l),
    update_root(RN, HList, NID-r),
    !,
    retcom(NID, OutPat, NList, T, N, RHS).
```

Building rules is simply accomplished by writing a new rule structure:

```
build_rule(OutPat, PrevList, N, RHS) :-
    assertz( rul(N, OutPat, RHS) ).
```

The **check_root** predicate is the first one that must deal with the pattern matching problem mentioned earlier. It covers three cases:

- There is no existing root which matches the term using Prolog's pattern matching. In this case a new root is added.
- There is an existing root that matches the term, atom for atom, variable for variable. In this case no new root is needed.
- There is no precise match and a new root is created.

In Prolog:

```
check_root(NID, Pattern, []) :-
    not(root(_, Pattern, _)),
    gen_nid(NID),
    assertz( root(NID, Pattern, []) ),
    !.
check_root(N, Pattern, List) :-
    asserta(temp(Pattern)),
    retract(temp(T1)),
    root(N, Pattern, List),
    root(N, T2, _),
    comp_devar(T1, T2),
    !.
check_root(NID, Pattern, []) :-
    gen_nid(NID),
    assertz( root(NID, Pattern, []) ).
```

The first clause is straight forward. The **gen_nid** predicate is used to generate unique numbers for identifying nodes in the Rete network.

The second clause is the difficult one. The first problem is to make a copy of **Pattern**, which Prolog will not unify with the original term. The easiest way is to assert the term and then retract it using a different variable name, as the first two goals of the second clause do. We now have both **Pattern** and **T1** as identical terms, but Prolog doesn't know they are the same and will not bind the variables in one when they are bound in the other.

We can now use **Pattern** to find the root which matches it, using Prolog's match. Again, not wishing to unify the variables, we call the root again using just the root identification. This gives us **T2**, the exact pattern in the root before unification with **Pattern**.

Now we have **T1** and **T2**, two terms which we know will unify in Prolog. The problem is to see if they are also identical in their placement of variables. For this we call **comp_devar**, which compares two terms after unifying all of the variables with generated strings.

A very similar procedure is used for **check_node**. It is a little more complex in that it needs to build and return the tokens that are the two inputs to a node. The arguments to **check_node** are:

- PNode** — token list from previous node
- PList** — list of successor nodes from previous node
- H** — new token being added
- HList** — successor nodes from root for token H
- NID** — returned ID of the node
- OutPat** — returned tokenlist from the node
- NList** — returned list of successor nodes from the node

The clauses for **check_node** are:

```

check_node(PNode, PList, H, HList, NID, OutPat, []) :-
    not (bi(_, PNode, H, _)),
    append(PNode, H, OutPat),
    gen_nid(NID),
    assertz( bi(NID, PNode, H, []) ),
    !.
check_node(PNode, PList, H, HList, NID, OutPat, NList) :-
    append(PNode, H, OutPat),
    asserta(temp(OutPat)),
    retract(temp(Tot1)),
    bi(NID, PNode, H, NList),
    bi(NID, T2, T3, _),
    append(T2, T3, Tot2),
    comp_devar(Tot1, Tot2),
    check_node(PNode, PList, H, HList, NID, OutPat, []) :-
        append(PNode, H, OutPat),
        gen_nid(NID),
        assertz( bi(NID, PNode, H, []) ).

```

The update predicates check to see if the new node is on the list of links from the old node. If so, nothing is done. Otherwise a new link is added by putting the new node id on the list.

```

update_root(RN, HList, NID) :-
    member(NID, HList),
    !.
update_root(RN, HList, NID) :-
    retract( root(RN, H, HList) ),
    asserta( root(RN, H, [NID|HList]) ).

```

```

update_node(root(RN), HList, NID) :-
    update_root(RN, HList, NID),
    !.
update_node(X, PrevList, NID) :-
    member(NID, PrevList),
    !.
update_node(PNID, PrevList, NID) :-
    retract( bi(PNID, L, R, _) ),
    asserta( bi(PNID, L, R, [NID|PrevList]) ).

```

The **comp_devar** predicate takes each term it is comparing, and binds all the variables to generated terms.

```

comp_devar(T1, T2) :-
    del_variables(T1),
    del_variables(T2),
    T1 = T2.

```

The **del_variables** predicate is used to bind the variables. The function that generates atoms to replace the variables is initialized the same way each time it is called, so if **T1** and **T2** have the same pattern of variables, they will be replaced with the same generated atoms and the terms will be identical.

```

del_variables(T) :-
    init_vargen,
    de_vari(T).

```

The basic strategy is to recognize the various types of structures and break them into smaller components. When a component is identified as a variable, it is unified with a generated atom.

First, **de_vari** looks at the case where the terms are lists. This is used for comparing token lists in **check_node**. It is a standard recursive predicate, which removes the variables from the head of the list and recursively calls itself with the tail. Note that unification will cause all occurrences of a variable in the head of the list to be bound to the same generated atom. The third clause covers the case where the term was not a list.

```

de_vari([]).
de_vari([H|T]) :-
    de_var(H),
    de_vari(T).
de_vari(X) :-
    de_var(X).

```

The first clause of **de_var** removes the time stamps from consideration. The next two clauses recognize the full frame pattern structure, and the attribute-value pair structure respectively.

```

de_var(X/_ ) :-
    de_var(X).
de_var(X-Y with List) :-
    de_v(X-Y),
    de_vl(List),
    !.
de_var(X-Y) :-
    de_v(X-Y),
    !.

```

The next predicates continue to break the structures apart until finally **d_v** is given an elementary term as an argument. If the term is a variable, an atom is generated to replace it. Otherwise the term is left alone. Due to Prolog's unification, once one instance of a variable is unified to a generated term, all other instances of that variable are automatically

unified to the same generated term. Thus, the generated atoms follow the same pattern as the variables in the full term.

```
de_vl([]).
de_vl([H|T]) :-
    de_v(H),
    de_vl(T).

de_v(X-Y) :-
    d_v(X),
    d_v(Y).

d_v(V) :-
    var(V),
    var_gen(V),
    !.
d_v(_).
```

The next two predicates are used to generate the variable replacements. They are of the form '#VAR_N', where N is a generated integer. In this case two built-in predicates of AAIS Prolog are used to convert the integer to a string and concatenate it to the rest of the form of the variable. The same effect could be achieved with the standard built-in, **name**, and a list of ASCII characters representing the string.

```
init_vargen :-
    abolish(varg, 1),
    asserta(varg(1)).

var_gen(V) :-
    retract(varg(N)),
    NN is N + 1,
    asserta(varg(NN)),
    int2string(N, NS),
    stringconcat("#VAR_", NS, X),
    name(V, X).
```

The system only handles simple rules so far, and does not take into account either negations or terms that are tests, such as comparing variables or other Prolog predicate calls.

Nodes to cover tests are easily added. They are very similar to the normal two-input nodes, but do not have memory associated with them. The left side is a token list just as in the two-input nodes. The right side has the test pattern. If a token passes the test, a new token with the test added is passed through the network.

The negated patterns store a count with each pattern in the left memory. That count reflects the number of right memory terms which match the left memory term. Only when the count is zero, is a token allowed to pass through the node.

8.6 Integration with Foops

Only a few changes have to be to *Foops* to incorporate the Rete network developed here.

A compile command is added to the main menu that builds the Rete network. It is called after the load command.

The commands to update working memory are modified to propagate tokens through the Rete network. This means calls to **addrete** and **delrete** as appropriate.

The refraction logic, which ensured the same instantiation would not be fired twice, is removed since it is no longer necessary.

The predicate that builds the conflict set is removed, since the conflict set is maintained by the predicates that process the network. The predicates that sort the conflict set are still used to select a rule to fire.

8.7 Design Tradeoffs

There are a number of design tradeoffs made in this version. The first is the classic space versus speed tradeoff. At each Rete memory, a full copy of each token is saved. This allows it to be retrieved and matched quickly during execution. Much space could be saved by only storing pointers to the working memory elements. These would have to be retrieved and the token reconstructed when needed.

The other tradeoff is with the flexibility of the frame system. With the frames in the original *Foops*, the frame patterns in the rules could take advantage of inheritance and general classes, as in the rules which determined a need for electrical plugs in a room. Since *Rete-Foops* instantiates the patterns before propagating tokens through the network, this does not work. This feature could be incorporated but would add to the burden and complexity of maintaining the network.

Exercises

- 8.1 Implement nodes which support rules which have tests such as $X > Y$, and negated frame patterns.
- 8.2 The implementation described in this chapter makes heavy use of memory by storing the full tokens in left and right memories. Build a new system in which space is saved by storing a single copy of the token and having pointers to it in left and right memory. The stored tokens just match single frame patterns. The complex tokens in the middle of the network contain lists of pointers to the simple tokens.
- 8.3 Experiment with various size systems to see the performance gains of the Rete version of *Foops*.
- 8.4 Figure out a way to allow *Rete-Foops* to use inheritance in frame patterns. That is, fix it so the rule which finds electric plugs works.
- 8.5 Build an indexed version of *Clam* and make performance experiments with it.

9 User Interface

The user interface issues for expert system shells can be divided between two classes of users and two different levels. The two users are the developer and the end-user of the application. The levels are the external interface and the internal function.

For the developer, the internal function must be adequate before the external interface becomes a factor. To build a configuration application, an easy-to-use backward chaining system is not as good as a hard-to-use forward chaining system. To build a large configuration system, an easy-to-use, low performance forward chaining system is not as good as a hard-to-use, high performance forward chaining system.

The same is true for the end-user. While there is increasing awareness of the need for good external interfaces, it should not be forgotten that the internal function is still the heart of an application. If a doctor wants a diagnostic system to act as an intelligent assistant and instead it acts as an all knowing guru, then it doesn't matter how well the external interface is designed. The system will be a failure. If a system can save a company millions of dollars by more accurately configuring computers, then the system will be used no matter how poor the external interface is.

Given that a system meets the needs of both the developer and the end-user, then the external interface becomes an essential ingredient in the satisfaction with the system. The systems developed so far have used a command driven, dialog type user interface. Increasingly windows, menus, and forms are being used to make interfaces easier to understand and work with. This chapter describes how to build the tools necessary for developing good user interfaces.

9.1 Object Oriented Window Interface

One of the major difficulties with computer languages in general, and Prolog in particular, is the lack of standards for user interface features. There are emerging standards, but there is still a long way to go. The windowing system described here includes a high-level, object oriented interface for performing common window, menu and form operations, which can be used with any Prolog or computer. The low level implementation will vary from Prolog to Prolog, and computer to computer.

The interface is object-oriented in that each window, menu and form in the system is considered to be a "window-object" that responds to messages requesting various behaviors. For example, a display window would respond to "write" messages, and both a menu and prompt window would respond to a "read" message.

The developer using the system defines window-objects to be used in an application and then uses a single predicate, **window**, to send messages to them. The system can be easily extended to include new messages and window-objects. For example, graphics can be included for systems that support it.

9.2 Developer's Interface to Windows

The windows module provides for overlapping windows of four basic flavors.

display — an output only window. The user may scroll the window up and down using the cursor keys.

menu — a pop-up vertical menu.

form — a fill-in-the-blanks form.

prompt — a one line input window.

The programmer creates the various windows by specifying their attributes with a create message. Other window messages are used to open, close, read or write the window.

All of the window operations are performed with the predicate **window**. It can either be specified with two or three arguments, depending on whether the message requires an argument list. The arguments are:

- the window-object name or description,
- the operation to be performed (message),
- the arguments for the operation (optional).

For example, the following Prolog goals cause a value to be selected from a main menu, a value to be written to a display window, and a useless window to be closed:

```
window(main_menu, read, X).
window(listing, write, 'Hello').
window(useless, close).
```

A window description is a list of terms. The functors describe the attribute, and the arguments are the value(s). Some of the attributes used to define a window are:

type(T) — type of window (display, prompt, menu, or form),

coord(R1, C1, R2, C2) — the upper left and lower right coordinates of useable window space,

border(Color) — the border color,

contents(Color) — the color of the contents of the window.

The following two attributes are used to initialize menus and forms:

menu(List) — List is a list of menu items.

form(Field_List) — Field_List defines the fields in the form. The field may be either a literal or a variable. The two formats are:

```
lit(Row:Col, Literal),
var(FieldName, Row:Col, Length, InitialValue).
```

Some examples of window descriptions are:

```
[type(display), coord(2, 3, 10, 42), border(white:blue),
 contents(white:blue)]
[type(menu), coord(10, 50, 12, 70), border(bright:green),
 menu(['first choice',
       'second choice',
       'third choice',
       'fourth choice'])]
[type(form), coord(15, 34, 22, 76), border(blue),
 form([lit(2:2, 'Field One'),
       var(one, 2:12, 5, ''),
       lit(4:2, 'Field Two'),
       var(two, 4:12, 5, 'init')])] ]
```

The first argument of each window command refers to a window-object. It may either be the name of a created window or a window description. If a description is used, the window is created and used only for the duration of the command. This is useful for pop up menus, prompts and forms. Created windows are more permanent.

Some of the messages which can be sent to windows are:

window(W, create, Description) — stores the Description with the name W;

window(W, open) — opens a window by displaying it as the current top window (usually not necessary since most messages open a window first);

window(W, close) — closes the window by removing the viewport from the screen while preserving the contents (for later re-opening);

window(W, erase) — closes the window, and erases the contents as well;

window(W, display, X) — writes the term X to the window.

To use the windows to improve the user interface of a simple expert system shell, the main menu can be made a pop-up menu. The text for questions to the user can appear in one window, and the user can respond using pop-up menus or prompt windows. The advice from the system can appear in another window.

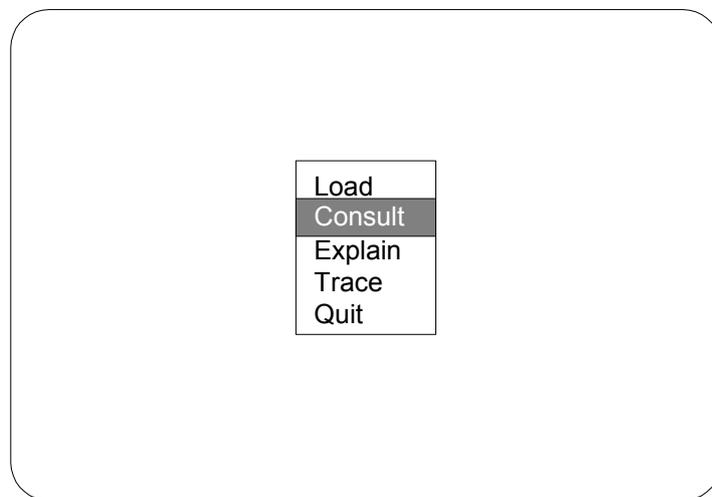


Figure 9.1 Main menu

First, the permanent windows are created during the initialization phase. The descriptions are stored in Prolog's database with the window name for use when the window is referenced. The windows for a simple interface include the main menu, the window for advice, and the window for questions to the user:

```
window_init:-
  window(wmain, create,
    [type(menu), coord(14, 25, 21, 40),
     border(blue), contents(yellow),
     menu(['Load',
          'Consult',
          'Explain',
          'Trace',
          'Quit'])]),
  window(advice, create,
    [type(display), coord(1, 1, 10, 78),
     border(blue:white), contents(blue:white)]),
  window(quest, create,
    [type(display), coord(13, 10, 13, 70),
     border(blue:white), contents(blue:white)]).
```

The main loop then uses the main menu:

```
go :-
  repeat,
```

```

window(wmain, read, X),
do(X),
fail.

```

The user sees a menu as in figure 9.1. The cursor keys move the highlighting bar, and the enter key selects a menu item. The **ask** and **menuask** predicates in the system use the windows to get information from the user. First **ask** writes the text of the question to the quest window, and then generates a pop-up prompt:

```

ask(A, V) :-
window(quest, write, A),
window([type(prompt), coord(16, 10, 16, 70),
border(white:blue),
contents(white:blue)]),
read, [' ', Y]),
asserta(known(A, Y)),
...

```

The **menuask** predicate also writes the text of the question to the quest window, but then dynamically builds a pop-up menu, computing the size of the menu from the length of the menu list:

```

menuask(Attribute, AskValue, Menu):-
length(Menu, L),
R1 = 16,
R2 is R1 + L - 1,
window(quest, write, Attribute),
window([type(menu), coord(R1, 10, R2, 40),
border(white:blue),
contents(white:blue), menu(Menu)]),
read, Value),
asserta(known(Attribute, Value)),
...

```

Figure 9.2 shows how a simple window interface would look with the Birds expert system.

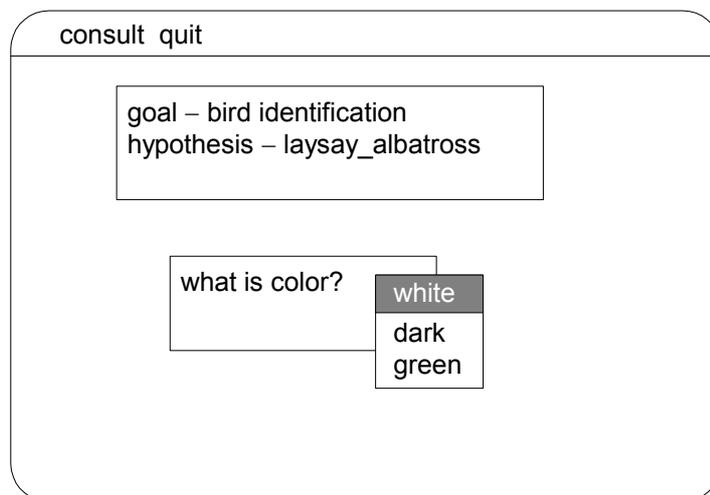


Figure 9.2 Window interface for dialog with Birds system

9.3 High-Level Window Implementation

The window module is constructed in layers. The top layer can be used with any Prolog. The lower layers have the actual implementations of the windows and vary from system to

system. The detailed examples will come from a Macintosh based Prolog (AAIS) using a rich user interface toolbox, and a PC based Prolog (Arity) using simple screen primitives.

Message Passing

At the top level, the interface is object oriented. This means messages are sent to the individual windows. One feature of object oriented systems is that messages are dispatched at run time based on the class of object. For example, the read message applies to both the prompt windows and the menu windows, but the implementation is different in each case. The implementations are called methods. The window predicate must determine what type of window is receiving the message, and what the correct method to call is:

```
window(W, Op, Args):-
    get_type(W, T),
    find_proc(T, Op, Proc),
    P =.. [Proc, W, Args],
    call(P),
    !.
```

The **get_type** predicate finds the type of the window, **find_proc** gets the correct method to call, and univ (=..) is used to call it.

When **window** is called with a window description as the first argument, it creates a temporary window, sends the message to it, and then deletes it. A two argument version of window is used to capture calls with no arguments.

```
window([H|T], Op, Args):-
    window(temp_w, create, [H|T]),
    window(temp_w, Op, Args),
    window(temp_w, delete),
    !.
window(W, Op) :-
    window(W, Op, []).
```

The **get_type** predicate uses **select_parm** to find the value for the type attribute of the window. It uses the stored window definition.

```
get_type(W, X):-
    select_parm(W, [type(X)]),
    !.
```

Window definitions are stored in structures of the form:

```
wd(W, AttributeList).
```

Inheritance

Another feature of object oriented systems is inheritance of methods. The objects are arranged in a class hierarchy, and lower level objects only have methods defined for them that are different from the higher level methods. In the window program, **type(window)** is the highest class, and the other types are subclasses of it. A predicate such as **close** is only defined for the window superclass and not redefined for the subclasses.

This makes it easy to add new window types to the system. The new types can inherit many of the methods of the existing types.

The classes are represented in Prolog using a subclass predicate:

```

subclass(window, display).
subclass(window, menu).
subclass(window, form).
subclass(window, prompt).

```

The methods are associated with classes by a method predicate. Some of the defined methods are:

```

method(window, open, open_w).
method(window, close, close_w).
method(window, create, create_w).
method(window, display, display_w).
method(window, delete, delete_w).
method(display, write, write_d).
method(display, writeline, writeline_d).
method(menu, read, read_m).
method(form, read, read_f).
method(prompt, read, read_p).

```

The **find_proc** predicate is the one that looks for the correct method to call for a given message and a given window type.

```

find_proc(T, Op, Proc):-
    find_p(T, Op, Proc),
    !.
find_proc(T, Op, Proc):-
    error([Op, 'is illegal operation for a window of type', T]).
find_p(T, Op, Proc):- method(T, Op, Proc),
    !.

find_p(T, Op, Proc):-
    subclass(Super, T),
    !,
    find_p(Super, Op, Proc).

```

This completes the definition of the high level interface, with the exception of the one utility predicate, **select_parm**. It is used by **get_type** to find the value of the type attribute, but is also heavily used by the rest of the system to find attributes of windows, such as coordinates. It has the logic built into it to allow for calculated attributes, such as height, and attribute defaults. It is called with a request list of the desired attributes. For example, to get a window's coordinates, height, and color:

```

select_parm(W, [coord(R1, C1, R2, C2), height(H),
               color(C)]).

```

The **select_parm** predicate gets the windows attribute list, and calls the **fulfill** predicate to unify the variables in the request list with the values of the same attributes in the window definition.

```

select_parm(W, RequestList):-
    wd(W, AttrList),
    fulfill(RequestList, AttrList),
    !.

```

The **fulfill** predicate recurses through the request list, calling **w_attr** each time to get the value for a particular attribute.

```

fulfill([], _):- !.
fulfill([Req|T], AttrList):-
    w_attr(Req, AttrList),
    !,
    fulfill(T, AttrList).

```

The `w_attr` predicate deals with three cases. The first is the simple case that the requested attribute is on the list of defined attributes. The second is for attributes that are calculated from defined attributes. The third sets defaults when the first two fail. Here are some of the `w_attr` clauses:

```
w_attr(A, AttrList):-
    member(A, AttrList),
    !.
% calculated attributes
w_attr(height(H), AttrList):-
    w_attr(coord(R1, _, R2, _), AttrList),
    H is R2 - R1 + 1,
    !.
w_attr(width(W), AttrList):-
    w_attr(coord(_, C1, _, C2), AttrList),
    W is C2 - C1 + 1,
    !.
% default attributes
w_attr(title(''), _).
w_attr(border(white), _).
w_attr(contents(white), _).
w_attr(type(display), _).
```

9.4 Low-Level Window Implementation

In addition to being a powerful language for artificial intelligence applications, Prolog is good at implementing more standard types of programs as well. Since most programs can be specified logically, Prolog is a very efficient tool. While we will not look at all the details in this chapter, a few samples from the low-level implementation should demonstrate this point. An entire overlapping window system with reasonable response time was implemented 100% in Prolog using just low-level screen manipulation predicates.

The first example shows predicates that give the user control over a menu. They follow very closely the logical specification of a menu driver. A main loop, `menu_select`, captures keystrokes, and then there are a number of rules, `m_cur`, governing what to do when various keys are hit. Here is the main loop for the Arity Prolog PC implementation:

```
menu_select(W, X):-
    select_parm(W, [coord(R1, C1, R2, _), width(L), attr(A)]),
    tmove(R1, C1), % move the cursor to first item
    revideo(L, A), % reverse video first item
    repeat,
    keyb(_, S), % read the keyboard
    m_cur(S, Z, w(W, R1, R2, C1, L, A)), % usually fails
    !,
    Z = X.
```

Here are four of the menu rules covering the cases where: the down arrow (scan code of 80) is hit (highlight the next selection); the down arrow is hit at the bottom of the menu (scroll the menu); the home key (scan code of 71) is hit (go to the top); and the enter key (scan code of 28) is hit (finally succeed and select the item).

```
m_cur(80, _, w(W, R1, R2, C1, L, A)):- % down arrow
    tget(R, _),
    R < R2,
    normvideo(L, A),
    RR is R + 1,
    tmove(RR, C1),
    revideo(L, A),
    !,
    fail.
```

```

m_cur(80, _, w(W, R1, R2, C1, L, A)):- % bottom down arrow
    tget(R, _),
    R >= R2,
    normvideo(L, A),
    scroll_window(W, 1),
    tmove(R2, C1),
    revideo(L, A),
    !,
    fail.
m_cur(71, _, w(W, R1, R2, C1, L, A)):- % home key
    normvideo(L, A),
    scroll_window(W, top),
    tmove(R1, C1),
    revideo(L, A),
    !,
    fail.
m_cur(28, X, w(W, R1, R2, C1, L, A)):- % enter key
    tget(R, _),
    select_stat(W, curline, Line), % current line
    Nth is Line + R - R1,
    getnth(W, Nth, X),
    normvideo(L, A),
    !.

```

Here is some of the code that deals with overlapping windows. When a window is opened, the viewport, which is the section of the screen it appears in, is initialized. The system maintains a list of active windows, where the list is ordered from the top window to the bottom. In the case covered here, the window to be opened is already on the active list, but not on the top.

```

make_viewport(W):-
    retract(active([H|T])),
    save_image(H),
    split(W, [H|T], L1, L2),
    w_chkover(W, L1, _),
    append([W|L1], L2, NL),
    assert(active(NL)).

```

The **save_image** predicate stores the image of the top window for redisplay if necessary. The **split** predicate is a list utility, which splits the active list at the desired window name. Next, **w_chkover** decides if the window needs to be redrawn due to overlapping windows on top of it, and then a new active list is constructed with the requested window on top.

The **w_chkover** predicate recurses through the list of windows on top of the requested window, checking each one for the overlap condition. If any window is overlapping, then the requested window is redrawn. Otherwise nothing needs to be done.

```

w_chkover(W, [], no).
w_chkover(W, [H|T], Stat):-
    w_nooverlap(W, H),
    w_chkover(W, T, Stat).
w_chkover(W, _, yes):-
    restore_image(W),
    !.

```

An overlap is detected by simply comparing the coordinates of the two windows.

```

w_nooverlap(Wa, Wb):-
    select_parm(Wa, [coord(R1a, C1a, R2a, C2a)]),
    select_parm(Wb, [coord(R1b, C1b, R2b, C2b)]),
    (R1a > R2b + 2;
     R2a < R1b - 2;
     C1a > C2b + 2;
     C2a < C1b - 2),

```

!.

As opposed to the PC implementation, which requires coding to the detail level, the Macintosh implementation uses a rich toolbox of primitive building blocks. However, the complexity of the toolbox sometimes makes it more difficult to perform simple operations. For example, to make a new window in the PC version, all that is necessary is to save the window definition:

```
make_window(W, Def):-
    asserta( wd(W, Def)).
```

In the Macintosh version, a number of parameters and attributes must be set up to interface with the Macintosh environment. The code to create a new window draws heavily on a number of built-in AAIS Prolog predicates that access the Macintosh toolbox.

```
make_window(W, L) :-
    define(wd, 2),
    include([window, quickdraw, types, memory]),
    fulfill([coord(R1, C1, R2, C2), title(T), visible(V),
    procid(Pid), behind(B), goaway(G),
    refcon(RC)], L),
    new_record(rect, R),
    new_record(windowptr, WP),
    setrect(R, C1, R1, C2, R2),
    pname(T, Tp),
    newwindow(WP, R, Tp, V, Pid, B, G, RC, WPptr),
    asserta( wd(W, [wptr(WPptr)|L]) ).
```

Notice that the special Macintosh window parameters are easily represented using the window attribute lists of the generic windows. The example above has attributes for **goaway**, a box the user can click to make a window go away, and **refcon** for attaching more sophisticated functions to windows. The **select_parm** predicate has intelligent defaults set for each of these parameters so the user does not have to worry about specifying them.

```
w_attr(goaway(false), _).
w_attr(refcon(0), _).
```

The generic window interface we developed recognizes a few fundamental types of window. The Macintosh also has numerous window types. The **w_attr** predicate is used to calculate values for the Macintosh parameters based on the generic parameters. The user only sees the generic parameters. Internally, the Macintosh parameters are used. Here is how the internal routines get a value for the Macintosh based parameter **wproc**, which controls the type of box the window is drawn in:

```
w_attr(wproc(documentproc), L) :-
    akindof(text, L),
    !.
w_attr(wproc(dboxproc), L) :-
    akindof(dialog, L),
    !.
w_attr(wproc(plaininbox), L) :-
    akindof(graph, L),
    !.
```

The **akindof** predicate checks type against the inheritance specified by the **subclass** predicate defined earlier.

```
akindof(ST, L) :-
    w_attr(type(T), L),
    subsub(ST, T),
```

! .

```
subsub (X, X) .  
subsub (Y, X) :-  
    subclass (C, X) ,  
    subsub (Y, C) .
```

As more toolboxes for user interface functions become available, such as Presentation Manager, the low level portions of this window package can be modified to take advantage of them. At the same time the simple object oriented high level interface described earlier can be maintained for easy application development and portability.

Exercises

- 9.1 Implement object oriented windows on the Prolog system you use.
- 9.2 Add windowing interfaces to all of the expert system shells developed so far.
- 9.3 Add controls as a window type. These are display windows that use a graphical image to represent a number. The easiest graphical control to implement is a thermometer in a text based system (such as the IBM PC in text mode). The controls can also contain digital readouts, which is of course even easier to implement.
- 9.4 Active images are controls which automatically display the contents of some variable in a system. For example, in the furniture placement system it would be interesting to have four controls that indicate the available space on each of the four walls. They can be elegantly implemented using the attached procedure in the **add** slot of the frames. Whenever a new value is added, the procedure sends it to the control window. (Note that add is called during update of the slot in this implementation.)
- 9.5 In the windowing interface for the various shells, have trace information appear in a separate window.
- 9.6 Add graphics windows to the system if your version of Prolog can support it.
- 9.7 In the main control loop, recognize certain user actions as a desire to manipulate the windows. For example, function keys might allow the user to pop various windows to the top. This would enable the system to keep trace information in one window, which is overlapped by the main window. The user could access the other windows by taking the appropriate actions.

10 Two Hybrids

This chapter describes two similar expert systems, which were developed at Cullinet Software, a large software vendor for IBM mainframes, VAXes, and PCs. The systems illustrate some of the difficulties in knowledge base design and show the different features needed in two seemingly very similar systems.

Both expert systems were designed to set parameters for the mainframe database, IDMS/R, at a new user site. The parameters varied from installation to installation, and it was necessary to have an experienced field support person set them at the site. Since field support people are expensive, the expert systems were written to allow the customer to set the parameters, thus freeing the support person for more demanding tasks.

The first, CVGEN, set the system generation (sysgen) parameters for the run time behavior of the system. This included such parameters as storage pool sizes, logging behavior, and restart procedures. These parameters had a serious effect on the performance of the system, and needed to be set correctly based on each site's machine configuration and application mix.

The second, AIJMP, set all of the parameters that ran an automated installation procedure. This included parameters that determined which modules to include and how to build installation libraries. These parameters determined how the software would reside at the customer's site.

The systems were built using a variation of the pure Prolog approach described earlier in the book. The inferencing parts of the system were separated from the knowledge base. It was surprising to find that even with two systems as similar as these — they both set parameters, the shell for one was not completely adequate for the other.

10.1 CVGEN

Various shells available on the PC were examined when CVGEN was built, yet none seemed particularly well suited for this application. The main difficulty centered around the nature of the dialog with the user. To a large degree, the expertise a field support person brought to a site was the ability to ask the right questions to get information from the systems programmers at the site, and the ability to judge whether the answers were realistic.

To capture this expertise, the knowledge base had to be rich in its ability to represent the dialog with the user. In particular:

- The system was designed to be used by systems programmers who were technically sophisticated, but not necessarily familiar with the parameters for IDMS/R. This meant fairly lengthy prompts were needed in the dialog with the user.
- The input data had to be subjected to fairly complex validation criteria, which was often best expressed in additional sets of rules. A large portion of the field person's expertise was knowing what values made sense in a particular situation.
- The output of the system had to be statements, which were syntactically correct for IDMS/R. This meant the rules not only found values for parameters but built the statements as well.

The first objective of the system was to gather the data necessary to set the parameters by asking meaningful questions of the systems programmer. This meant providing prompts with a fair amount of text.

The next objective of the system was to validate the user's input data. The answers to the questions needed to be checked for realistic values. For example, when asking for the desired number of simultaneous batch users, the answer had to be checked for reasonableness based on the size of machine.

A similar objective was to provide reasonable default answers for most of the questions. As were the edit checks, the defaults were often based on the particular situation and required calculation using rules.

Given these objectives, the questioning facility needs to have the ability to call rule sets to compute the default before asking a question, and another rule set to validate the user's response. It also needs to be able to store questions that are up to a full paragraph of text.

The knowledge base needs to be designed to make it easy for the experts to view the dialog, and the edit and default rules. The knowledge base also needs some pure factual information.

The actual rules for inferencing were relatively simple. The system had a large number of shallow rules (the inference chains were not very deep), which were best expressed in backward chaining rules. The backward chaining was natural since the experts also tackled the problem by working backward from the goals of setting individual parameter values.

Also, since the system was setting parameters, uncertainty was not an issue. The parameter was either set to a value or it wasn't. For this reason pure Prolog was used for the main rule base.

Pure Prolog had the additional advantage of making it easy for the rules to generate IDMS/R syntax. The arguments to the parameter setting rules were lists of words in the correct syntax, with variables in the positions where the actual value of the parameter was placed. The rules then sought those values and plugged them into the correct syntax.

10.2 The Knowledge Base

The knowledge base is divided into six parts, designed to make it easy for the expert to examine and maintain it. These are:

- main rules for the parameters;
- rules for derived information;
- questions for the user;
- rules for complex validation;
- rules for complex default calculations;
- static information.

Rule for parameters

The rules for each parameter are stored in the knowledge base with the parameter name as the functor. Thus each parameter is represented by a predicate. The argument to the predicate is a list with the actual IDMS/R syntax used to set the parameter. Variables in the

list are set by the body of the predicate. A separate predicate, **parm**, is used to hold the predicate names that represent parameters.

Most knowledge bases are designed with askable information listed separately from the rules, as in the earlier examples in the book. In this case however, the experts wanted the relationship between user dialog and rules to be more explicit. Therefore the **ask** predicate is embedded in the body of a rule whenever it is appropriate.

In the following example, the parameter is **ina**, which when set will result in a text string of the form INACTIVE INTERVAL IS X, where X is some time value. Some of the subgoals, which are derived from other rules, are **online_components** and **small_shop**, whereas **int_time_out_problems** is obtained from the user.

```
parm(ina).

ina(['INACTIVE INTERVAL IS', 60]):-
    online_components,
    small_shop.
ina(['INACTIVE INTERVAL IS', 60]):-
    online_components,
    heavily_loaded.
ina(['INACTIVE INTERVAL IS', 60]):-
    ask(initial_install, no),
    online_components,
    ask(int_time_out_problems, yes).
ina(['INACTIVE INTERVAL IS', 30]):-
    online_components.
```

Some parameters also have subparameters that must be set. The structure of the knowledge base reflects this situation:

```
parm(sys).

sys(['SYSCTL IS', 'NO']):-
    never.
sys(['SYSCTL IS', 'SYSCTL']):-
    os_class(os).

subprm(sys, dbn, ['DBNAME IS', 'NULL']):-
    ask(initial_install, no),
    ask(multiple_dictionaries, yes),
    ask(db_name, null).
subprm(sys, dbn, ['DBNAME IS', V1]):-
    ask(initial_install, no),
    ask(multiple_dictionaries, yes),
    ask(db_name, V1),
    V1 \== null.
```

Rules for derived information

The next part of the knowledge base contains the level of rules below the parameter/subparameter level. These rules represent derived information. They read as standard Prolog. Here are a few examples:

```
heavily_loaded:-
    ask(heavy_cpu_utilization, yes),
    !.
heavily_loaded:-
    ask(heavy_channel_utilization, yes),
    !.
```

```

mvs_xa:-
  ask(operating_system, mvs),
  ask(xa_installed, yes),
  !.

online_components:-
  dc_ucf,
  !.
online_components:-
  ask(cv_online_components, yes),
  !.

```

Questions for the user

The next portion of the knowledge base describes the user interaction. Standard Prolog rules do not cover this case, so special structures are used to hold the information. Operator definitions are used to make it easy to work with the structure.

The first two examples show some of the default and edit rules, which are simple enough to keep directly in the question definition.

```

quest abend_storage_size
default 200
edit between( 0, 32767)
prompt
  ['Enter the amount of storage, in fullwords, available',
  'to the system for processing abends in the event',
  'of a task control element (TCE) stack overflow.',
  'Note that this resource is single threaded.'].
quest abru_value
default no
edit member( [yes, no])
prompt
  ['Do you want the system to write a snap dump to the',
  'log file when an external run unit terminates',
  'abnormally?'].

```

The next two rules require more complex edit and default rule sets to be called. The square brackets in the default field indicate there is a rule set to be consulted. In these examples, **ed_batch_user** will be called to check the answer to **allowed_batch_users**, and **def_storage_cushion** is used to calculate a default value for **storage_cushion_size**.

```

quest allowed_batch_users
default 0
edit ed_batch_user
prompt
  ['How many concurrent batch jobs may access',
  'the CV at one time?'].
quest storage_cushion_size
default [def_storage_cushion]
edit between( 0, 16384)
prompt
  ['How many bytes of storage cushion would',
  'you like? When available storage is less than the',
  'cushion no new tasks are started. A recommended',
  'value has been calculated for you.'].

```

Default rules

The next two sections contain the rules that are used for edit and default calculations. For example, the following rules are used to calculate a default value for the storage cushion

parameter. Notice that it in turn asks other questions and refers to the settings of another parameter, in this case the storage pool (**stopoo**).

```
def_storage_cushion(CUS):-
  ask(initial_install, yes),
  stopoo(L_, SP],
  PSP is SP / 10,
  min(PSP, 100, CUS),
  !.
def_storage_cushion(V1):-
  ask(total_buffer_pools, V2),
  stopoo(L_, V3]),
  ask(maximum_tasks, V4),
  V1 is (V2 + V3 + 3) / (3 * V4),
  !.
```

Rules for edits

Here are the rules that are used to edit the response to the number of batch users. The user's response is passed as the argument, and rules succeed or fail in standard Prolog fashion depending on the user's response.

```
ed_batch_user(V1):-
  V1 =< 2,
  !.
ed_batch_user(V1):-
  machine_size(large),
  V1 =< 10,
  !.
ed_batch_user(V1):-
  machine_size(medium),
  V1 =< 5,
  !.
ed_batch_user(V1):-
  machine_size(small),
  V1 =< 3,
  !.
```

Static information

The final section contains factual information. For example, here is a table of the MIPS ratings for various machines, and the rules used to broadly classify machines into sizes.

```
mac_mips('4381-1', 1.7).
mac_mips('4381-2', 2.3).
mac_mips('3083EX', 3.7).
mac_mips('3083BX', 6.0).
mac_mips('3081GX', 12.2).
mac_mips('3081KX', 15.5).
mac_mips('3084QX', 28.5).

mips_size(M, tiny):-
  M < 0.5,
  !.
mips_size(M, small):-
  M >= 0.5,
  M < 1.5,
  !.
mips_size(M, medium):-
  M >= 1.5,
  M < 10,
  !.
```

```
mips_size(M, large):-
    M >= 10.
```

The knowledge base is designed to reduce the semantic gap between it and the way in which the experts view the knowledge. The main parameter setting rules are organized by parameter and subparameter as the expert expects. The secondary rules for deriving information and the queries to the user are kept in separate sections.

The dialog with the user is defined by data structures, which act as specialized frames with slots for default routines and edit routines. Their definition is relatively simple since the frames are not general purpose, but designed specifically to represent knowledge as the expert describes it.

The standard Prolog rule format is used to define the edit and default rules. In the knowledge base the rules are simple, so Prolog's native syntax is not unreasonable to use. It would of course be possible to utilize a different syntax, but the Prolog syntax captures the semantics of these rules exactly. The experts working with the knowledge base are technically oriented and easily understand the Prolog syntax. Finally, supporting data used by the system is stored directly in the knowledge base.

It is up to the inference engine to make sense of this knowledge base.

10.3 Inference Engine

The inference is organized around the specialized knowledge base. The highest level predicates are set up to look for values for all of the parameters. The basic predicate **set_parms** accomplishes this. It uses the **parm** predicate to get parameter names and then uses the **univ** (=..) built-in function to build a call to a parameter setting predicate.

```
set_parms:-
    parm(Parm),
    set_parm(Parm),
    fail.
set_parms:-
    write('no more parms'), nl.
set_parm(Parm):-
    get_parm(Parm, Syntax),
    write(Parm), write(': '),
    print_line(Syntax), nl,
    subs(Parm).

get_parm(Parm, Syntax):-
    PS =.. [Parm, Syntax],
    call(PS),
    !.

subs(Parm):-
    subprm(Parm, Sub, Syntax),
    write(Parm), write('/'), write(Sub), write(':'),
    print_line(Syntax), nl,
    subs(Sub),
    fail.
subs(Parm):-
    true.
```

The next portion of the inference engine deals with the questions to the user. The following operator definitions are used to define the data structure for questions.

```
:-op(250, fx, quest).
:-op(240, yfx, default).
:-op(240, yfx, edit).
```

```
:-op(240, yfx, prompt).
```

The basic **ask** predicate follows the patterns used earlier, but is more complex due to the fact that it handles both attribute-value pairs and object-attribute-value triples. The implementation of triples is relatively straight-forward and not worth repeating. The interesting portions of **ask** have to do with handling defaults and edits.

The following code is used by the **ask** predicate to perform edits on a user response. It is called after the user enters a value. If the edit fails, the user is presented with an explanation for why the edit failed, and is reprompted for the answer.

The third argument to **edit** is the edit criterion. It could be a simple edit such as **member** or **less_than**, or one of the more complex edit rules. The built-in **univ** (=..) is used to construct the goal that is called for the edit process. The actual code is slightly more complex due to additional arguments holding trace information for explanations.

```
edit(X, X, none):- !. % passes, no edit criteria.
edit(X, X, Ed) :-
    Ed =.. [Pred | Args],
    Edx =.. [Pred, X | Args],
    call(Edx),
    !.
edit(X, X, not(Ed)):-
    Ed =.. [Pred | Args],
    Edx =.. [Pred, X | Args],
    notcall(Edx),
    !.
```

The default is handled in a similar fashion. It is calculated before the prompt to the user, and is displayed in the answer window. Just hitting enter allows the user to take the default rather than entering a new value.

```
default([], []):- !.
default(D, D):-
    atomic(D),
    !.
default([D], X):-
    P =.. [D, X],
    call(P).
```

10.4 Explanations

Explanations become a bit more difficult with the **ask** predicate. The how questions are handled pretty much as in the *Clam* and *Native* systems described earlier in the book. Since why traces require overhead during the inference process, and performance is a key issue for a system with a long dialog such as this one, the why trace implementation is different from that in *Native*. The basic strategy is to use pure Prolog as indicated for most of the inferencing, but to redo the inference using a Prolog in Prolog inference engine to answer why questions.

In order to do this the system must in fact restart the inference, but since the parameters are all basically independent, the why trace need only restart from the last call to set a parameter. For this reason, the **set_parm** predicate writes a record to the database indicating which parameter is currently being set.

Once the why trace gets into **ask**, the Prolog in Prolog must stop. In fact, the question might have arisen from setting a parameter, calculating a default value or specifying an edit criteria. Again, for these cases a flag is kept in the database so that trace knows the current situation.

The why trace then starts at the beginning, traces pure Prolog inferencing until it encounters **ask**. The why explanation then notes that it is in **ask**, and finds out from the database if **ask** has gone into either **default** or **edit**. If so it proceeds to trace the **default** or **edit** code. The final explanation to the user has the Prolog traces interspersed with the various junctions caused by **edit** and **default** in **ask**.

This system is a perfect example of one in which the explanations are of more use in diagnosing the system than in shedding light on an answer for the user. Many of the rules are based solely on empirical evidence, and reflect no understanding of underlying principles. For this reason a separate explanation facility was added to the knowledge base that would explain in English the rationale behind the setting of a particular parameter.

For example, the setting of the **maxeru** parameter is relatively complex. The rule, while correct in figuring a value for the parameter, does not give much insight into it. The separate **exp** predicate in the knowledge base is displayed in addition to the rule if the user asks how a value of **maxeru** was derived.

```
parm(maxeru) .

maxeru( ['MAXIMUM ERUS IS', MAXERU]):-
    maxeru_potential(PMERU),
    max_eru_tas(F),
    MAXERUF is PMERU * F,
    MAXERU is integer(MAXERUF),
    explain(maxerutas01) .

exp(maxerutas01,
    ['MAXERUS and MAXTASKS are set together. They are both ',
    'potentially set to values which are dictated by the size ',
    'of the terminal network. The total tasks for both is then ',
    'compared to the maximum realistic number for the ',
    'machine size. If the total tasks is too high, both ',
    'MAXERUS and MAXTASKS are scaled down ',
    'accordingly.']).
```

10.5 Environment

CVGEN is also designed to handle many of the details necessary in a commercially deployed system. These details include the ability to change an answer to a question, save a consultation session and restore it, build and save test runs of the system, and the ability to list and examine the cache and the knowledge base from within a consultation. The system also includes a tutorial, which teaches how to use the system.

Most of these features are straight-forward to implement, however changing a response is a bit tricky. When the user changes an answer to a question, it is almost impossible to predict what effects that will have on the results. Whole new chains of inferencing might be triggered. The safest way to incorporate the change is to rerun the inference. By saving the user's responses to questions, the system avoids asking any questions previously asked. New questions might be asked due to the new sequence of rules fired after the change.

The facts which are stored are not necessarily the same as the user's response. In particular, the user response of "take the default" is different from the actual answer, which is the default value itself. For this reason, both the facts and the user responses to questions are cached. Thus, when the user asks to change a response, the response can be edited and the inference rerun without reprompting for all of the answers.

This list of responses can also be used for building test cases, which are rerun as the knowledge base is modified.

10.6 AIJMP

The AIJMP system seemed on the surface to be identical to the CVGEN system. Both set parameters. It was initially assumed that the shell used for CVGEN could be applied to AIJMP as well. While this was in general true, there were still key areas that needed to be changed.

The differences have much to do with the nature of the user interaction. The CVGEN system fits very nicely into the classic expert system dialog as first defined in the MYCIN system. The system tries to reach goals and asks questions as it goes. However, for AIJMP there is often the need for large amounts of tabular data on various pieces of hardware and software. For these cases a question and answer format becomes very tedious for the user; a form-based front end to gather information is much more appropriate.

AIJMP uses forms to capture some data, and dialogs to ask for other data as needed. This led to the need to expand the basic inferencing to handle these cases.

Another difficulty became evident in the nature of the expertise. Much of what was needed was purely algorithmic expertise. For example, part of the system uses formulas to compute library sizes based on different storage media. Many of the parameters required both rules of thumb and algorithmic calculations.

The best solution to the problem, for the knowledge engineer, was to build into the inference engine the various predicates which performed calculations. This way they could be referred to easily from within the rules.

Some of the declarative knowledge required for AIJMP could not be easily represented in rules. For example, many products depend on the existence of co-requisite products. When the user enters a list of products to be installed, it must be checked to make sure all product dependencies are satisfied. The clearest way to represent this knowledge was with specialized data structures. Operators are used to make the structures easy to work with.

```
product 'ads batch 10.1'
psw [adsb]
coreqs ['idms db', 'i data dict'].
product 'ads batch 10.2'
psw [adsb]
coreqs ['idms db', 'i data dict'].
product 'ads online'
psw [adso, nlin]
coreqs ['idms db', 'idms cv', 'i data dict',
        'idms dc' / 'idms ucf'].
product auditor
psw [audi, culp]
coreqs [].
product autofile
psw [auto]
coreqs [].
```

The inference engine was enhanced to use this structure for co-requisite checking. The design goal is to make the knowledge base look as familiar as possible to the experts. With Prolog, it is not difficult to define specialized structures that minimize semantic gap and to modify the inference engine to use them.

One simple example of how the custom approach makes life easier for the expert and knowledge engineer is in the syntax for default specifications in the questions for the user. The manual on setting these parameters used the "@" symbol to indicate that a parameter had as its default the value of another parameter. This was a shorthand syntax well understood by the experts. In many cases the same value (for example a volume id on a

disk) would be used for many parameters by default. Only a slight modification to the code allowed the knowledge to be expressed using this familiar syntax:

```
quest loadunit
default @ diskunit
edit none
prompt
['What is the unit for the load library?'].
```

One of the major bottlenecks in expert system development is knowledge engineering. By customizing the knowledge base so it more closely matches the expert's view of the knowledge domain, the task becomes that much simpler. A simple change such as this one makes it easier for the expert and the knowledge base to interact.

10.7 Summary

These two systems show how some of the techniques in this book can be used to build real systems. The examples also show some of the difficulties with shells, and the advantages of customized systems in reducing semantic gap.

Exercises

- 10.1 Incorporate data structures for user queries with edits and defaults for the *Clam* shell.
- 10.2 The CVGEN user query behavior can be built into *Foops* when a value is sought from the frame instances. If there is no other way to get the value, the user should be queried. Additional facets can be used for prompt, default, and edit criteria, which the inference engine uses just like in CVGEN.
- 10.3 Add features of CVGEN to the shells that are needed for real world applications. These include the ability to save user responses, allow editing of responses, saving a consultation, and rerunning a consultation. The last feature is essential for testing and debugging systems. Old test runs can be saved and rerun as the knowledge base changes. Hopefully the changes will not adversely affect the old runs.

11 Prototyping

Regardless of whether one is going to use Prolog to build a finished application, Prolog is still a powerful tool for prototyping the application. The problem might fit nicely into *Clam* or *Foops*, in which case those systems should be used for the prototype. Otherwise, pure Prolog can be used to model the application.

In an expert system prototype it is important to model all of the different types of knowledge that will be used in the application. Initial knowledge engineering should be focused on what types of information the expert uses and how it is used. The full range of expertise should be modelled, but not to the depth required for a real system.

The Prolog rules used in a prototype can be quickly molded to get the desired effects in the application. The clean break between the inference engine and the knowledge base can be somewhat ignored to allow more rapid development of the prototype. Explanations, traces and many of the other features of an expert system are left out of the prototype. The I/O is implemented simply to just give a feeling for the user interaction with the system. The full system can be more elegantly designed once the prototype has been reviewed by the potential users.

11.1 The Problem

This section describes the building of a prototype system, which acts as an advisor for a mainframe software salesperson. A good sales person must not only be congenial and buy lunches, but must also have good product knowledge and know how to map that knowledge onto a potential customer's needs. The type of knowledge needed by the sales person is different from that typically held by a technical person.

The technical person thinks of a product in terms of its features, and implementation details. The sales person must think of the prospect's real and perceived needs and be able to map those to benefits provided by the features in the product. That is, the sales person must understand the prospect's objectives and be able to present the benefits and features of the product that help meet those objectives.

The salesperson must also have similar product knowledge about the competitor's products and know which benefits to stress that will show up the weaknesses in the competitor's product for the particular prospect.

In addition to this product knowledge, the sales person also has rules for deciding whether or not the prospect is likely to buy, and recognizing various typical sales situations.

With a large workload, it is often difficult for a sales person to keep up on product knowledge. An expert system, which helps the sales person position the products for the prospect, would be a big asset for a high tech sales person. The *Sales Advisor* system is a prototype of such a system, designed to help in the early stages of the sales cycle.

11.2 The Sales Advisor Knowledge Base

The ways in which sales people mentally organize product knowledge are fairly consistent. The knowledge base for the sales advisor should be organized in a format which is as close to the sales person's organization of the knowledge as possible. This way the semantic gap will be reduced and the knowledge base will be more easily maintained by a domain expert.

The main types of knowledge used by the salesperson fall into the following categories:

Qualification — the way in which the salesperson determines if the prospect is a good potential customer and worth pursuing;

Objective Benefit Feature (OBF) analysis — the way a salesperson matches the customer's objectives with the benefits and features of the product;

Competitive analysis — the way a salesperson decides which benefits and features to stress based on the competitor's weaknesses;

Situation analysis — the way a salesperson determines if the products will run in the prospect's shop.

Miscellaneous advice — various rules covering different situations, which do not fall neatly in the above categories.

Having this overall organization, we can now begin to prototype the system. The first step is to design the knowledge base. Simple Prolog rules can be used wherever possible. The knowledge for each area will be considered separately. The example uses the products sold for mainframe computers by Cullinet Software.

Qualifying

First we implement the knowledge for qualifying the prospect. This type of knowledge falls easily into a rule format. The final version will probably need some uncertainty handling as in *Clam*, but it is also important for this system to provide more text output than *Clam* provides. The quickest way to build the prototype is to use pure Prolog syntax rules with I/O statements included directly in the body of the rule. *Clam* can be used later with modifications for better text display.

Two examples of qualifying rules are: the prospect must have an IBM mainframe, and the prospect's revenues must be at least \$30 million. They are written as **unqualified** since if the prospect fails a test then it is unqualified.

```
unqualified:-
    not computer('IBM'),
    advise('Prospect must have an IBM computer'),
    nl.
unqualified:-
    revenues(X),
    X < 30,
    advise('Prospect is unlikely to buy IDMS with revenues under $30
million'),
    nl.
```

Objectives - Benefits - Features

Sales people typically store product knowledge in a tabular form called an objective-benefit-feature chart, or OBF chart. It categorizes product knowledge so that for each objective of the customer, the benefits of the product for meeting that objective and the features of the product are detailed.

For the prototype we can simplify the prospect objectives by considering three main ones: development of applications, building an information center and building efficient production systems. Each prospect might have a different one of these objectives. The benefits of each product in the product line varies for each of these objectives. This information is stored in Prolog structures of three arguments called **obf**. The first argument is the feature (or product), the second is the customer objective and the third is the benefit that is stressed to the prospect.

```

obf('IDMS/R', development,
    'IDMS/R separates programs from data, simplifying
development.').

obf('IDMS/R', information,
    'IDMS/R maintains corporate information for shared access.').

obf('IDMS/R', production,
    'IDMS/R allows finely tuned data access for optimal
performance.').

obf('ADS', development,
    'ADS automates many programming tasks thus increasing
productivity.').

obf('ADS', production,
    'ADS generates high performance compiled code.').

obf('OLQ', development,
    'OLQ allows easy validation of the database during
development.').

obf('OLQ', information,
    'OLQ lets end users access corporate data easily.').

obf('OLE', information,
    'OLE lets users get information with English language
queries.').

```

By using a chart such as this, the salesperson can stress only those features and benefits that meet the prospect's objectives. For example, OLE (OnLine English — a natural language query) would only be mentioned for an information center. OLQ (OnLine Query — a structured query language) would be presented as a data validation tool to a development shop, and as an end user query tool to an information center.

This knowledge could have been stored as rules of the form:

```

obf('OLE', 'OLE lets users get information in English') :-
    objective(information).

```

This type of rule is further away from the way in which the experts understand the knowledge. The structures are more natural to deal with and the inference engine can be easily modified to deal with what is really just a different format of a rule.

Situation Analysis

The next key area is making sure that the products are compatible with the customer's configuration. We wouldn't want to sell something that doesn't work. For example, OLE would not run at the time on a small machine or under a DOS operating system.

```

unsuitable('OLE') :-
    operating_system(dos).
unsuitable('OLE') :-
    machine_size(small).

```

Competitive Analysis

A good sales person will not directly attack the competition, but will use the competition's weakness to advantage. This is done by stressing those aspects of a product which highlight the competitor's weakness. That is, how can our product be differentiated from the competitor's. For example, two of Cullinet's main competitors were IBM and ADR.

Both IBM and Cullinet provided systems that performed well, but Cullinet's was easy to use, so ease of use was stressed when the competitor was IBM. ADR's system was also easy to use, but did not perform as well as Cullinet's, so against ADR performance was stressed.

```
prod_dif('IDMS/R', 'ADR',
        'IDMS/R allows specification of linked lists for high
performance.').

prod_dif('IDMS/R', 'IBM',
        'IDMS/R allows specification of indexed lists for easy
access.').

prod_dif('ADS', 'ADR',
        'ADS generates high performance code.').

prod_dif('ADS', 'IBM',
        'ADS is very easy to use.').
```

Miscellaneous Advice

Besides this tabular data, there are also collections of miscellaneous rules for different situations. For example, there were two TP monitors, UCF, and DC. One allowed the user to use CICS for terminal networks, and the other provided direct control of terminals. The recommendation would depend on the situation. Another example is dealing with federal government prospects, which required help with the Washington office as well. Another rule recommends a technical sales approach, rather than the business oriented sell, for small shops that are not responding well.

```
advice:-
  not objective(production),
  tp_monitor('CICS'),
  online_applications(many),
  nl,
  advise('Since there are many existing online applications and'),
  nl,
  advise('performance isn''t an issue suggest UCF instead of DC'),
  nl.
advice:-
  industry(government),
  government(federal),
  nl,
  advise('If it's the federal government, make sure you work'),
  nl,
  advise(' with our federal government office on the account'),
  nl.
advice:-
  competition('ADR'),
  revenues(X),
  X < 100,
  friendly_account(no),
  nl,
  advise(' Market database technical issues'),
  nl,
  advise(' Show simple solutions in shirt sleeve sessions' ),
  nl.
```

User Queries

Finally, the knowledge base contains a list of those items which will be obtained from the user.

```

competition(X):-
    menuask('Who is the competition?',
            X, ['ADR', 'IBM', 'other']).

computer(X):-
    menuask('What type of computer are they using?',
            X, ['IBM', 'other']).

friendly_account(X):-
    menuask('Has the account been friendly?',
            X, [yes, no]).

government(X):-
    menuask('What type of government account is it?',
            X, [federal, state, local]).

industry(X):-
    menuask('What industry segment?',
            X, ['manufacturing', 'government', 'other']).

machine_size(X):-
    menuask('What size machine are they using?',
            X, [small, medium, large]).

objective(X):-
    menuask('What is the main objective for looking at DBMS?',
            X, ['development', 'information', 'production']).

online_applications(X):-
    menuask('Are there many existing online applications?',
            X, [many, few]).

operating_system(X):-
    menuask('What operation system are they using?',
            X, ['OS', 'DOS']).

revenues(X):-
    ask('What are their revenues (in millions)?',X).

tp_monitor(X):-
    menuask('What is their current TP monitor?',
            X, ['CICS', 'other']).

```

11.3 The Inference Engine

Now that a knowledge base has been designed, which has a reasonably small semantic gap with the expert's knowledge, the inference engine can be written. For the prototype, some of the knowledge is more easily stored in the inference engine. The high level order of goals to seek is stored in the main predicate, **recommend**.

```

recommend:-
    qualify,
    objective_products,
    product_differentiation,
    other_advice,
    !.
recommend.

```

First, the prospect is qualified. The **qualify** predicate checks to make sure there are no unqualified rules which fire.

```

qualify:-
    unqualified,
    !,

```

```
fail.  
qualify.
```

The **objective_products** predicate uses the user's objectives and the OBF chart to recommend which products to sell and which benefits to present. It makes use of the **unsuitable** rules to ensure no products are recommended, which will not work in the customer's shop.

```
objective_products:-  
    objective(X),  
    advise('The following products meet objective'),  
    advise(X), nl, nl,  
    obf(Product, X, Benefit),  
    not unsuitable(Product),  
    advise(Product:Benefit),nl,  
    fail.  
objective_products.
```

Next, the product differentiation table is used in a similar fashion.

```
product_differentiation:-  
    competition(X),  
    prod_dif(_,X,_),  
    advise('Since the competition is '),  
    advise(X),  
    advise(', stress:'), nl, nl,  
    product_diff(X),  
    !.  
product_differentiation.  
  
product_diff(X):-  
    prod_dif(Prod, X, Advice),  
    tab(5), advise(Advice), nl,  
    fail.  
product_diff(_).
```

Finally, the other advice rules are all tried.

```
other_advice:-  
    advice,  
    fail.  
other_advice.
```

11.4 User Interface

For a prototype, the user interface is still a key point. The system will be looking for supporters inside an organization, and it must be easy for people to understand the system. The windowing environment makes it relatively easy to put together a reasonable interface.

For this example, one display window is used for advice near the top of the screen, and a smaller window near the bottom is used for questions to the user. Pop-up menus and prompter windows are used to gather information from the user. Figure 11.1 shows the user interface.

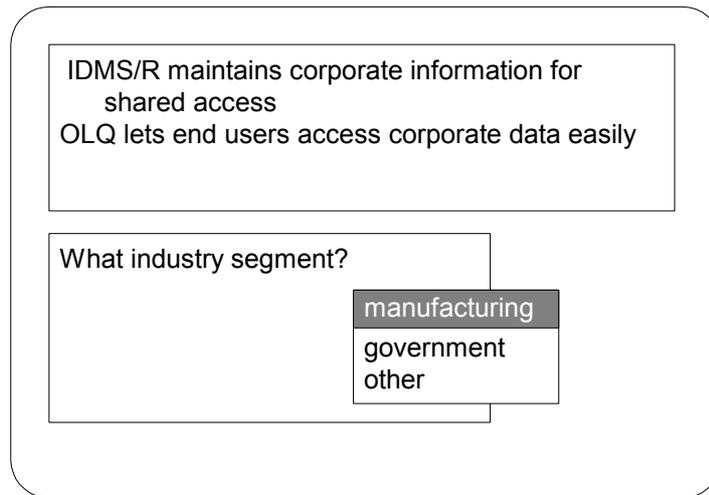


Figure 11.1 User interface of sales advisor prototype

The two display windows are defined at the beginning of the session.

```

window_init:-
    window(advice, create, [type(display), coord(1,1,10,78),
                           border(blue:white),
                           contents(blue:white)]),
    window(quest, create, [type(display), coord(13,10,13,70),
                           border(blue:white),
                           contents(blue:white)]).

```

The prompt and pop-up menu windows are defined dynamically as they are needed. The **ask** and **menuask** predicates work as in other examples. Here are the clauses that interface with the user.

```

ask(A,V):-
    window(quest,write,A),
    window([type(prompt),coord(16,10,16,70),border(white:blue),
           contents(white:blue)],
           read,['',Y]),
    asserta(known(A,Y)),
    Y = V.

menuask(Attribute,AskValue,Menu):-
    length(Menu,L),
    R1 = 16,
    R2 is R1 +L - 1,
    window(quest,write,Attribute),
    window([type(menu),coord(R1,10,R2,40),border(white:blue),
           contents(white:blue),menu(Menu)],
           read,AnswerValue),
    asserta(known(Attribute,AnswerValue)),
    AskValue = AnswerValue.

```

The **advise** predicate uses the predefined display window, **advice**.

```

advise([H|T]):-
    window(advice,writeline,[H|T]),
    !.
advise(X):-
    window(advice,write,X).

```

11.5 Summary

One can model a fairly complex domain relatively quickly in Prolog, using the tools available. A small semantic gap on the knowledge base and good user interface are two very important points in the prototype.

Exercises

- 11.1 Prototype an expert system that plays poker or some similar game. It will need to be specialized to understand the particular knowledge of the game. Experiment with the prototype to find the best type of user interface and dialog with the system.

12 Rubik's Cube

This chapter describes a Prolog program that solves Rubik's cube. The program illustrates many of the knowledge engineering problems in building expert systems. Performance is a key issue and affects most of the design decisions in the program.

This program differs from the others in the book in that the knowledge and the reasoning are all intertwined in one system. The system uses Prolog's powerful data structures to map the expertise for solving a cube into working code. It illustrates how to build a system in a problem domain that does not fit easily into the attribute-value types on data representation used for the rest of the book.

Like most expert systems, the program can perform at a level comparable to a human expert, but does not have an "understanding" of the problem domain. It is simply a collection of the rules, based on *Unscrambling the Cube* by Black & Taylor, that an expert uses to solve the cube. Depending on the machine, it unscrambles cubes as fast or faster than a human expert. It does not, however, have the intelligence to discover the rules for solving Rubik's cube from a description of the problem.

A Rubik's cube program illustrates many of the trade-offs in AI programs. The design is influenced heavily by the language in which the program is written. The representation of the problem is the key, but each language provides different capabilities for knowledge representation and tools for manipulating the knowledge.

Performance has always been the issue with expert systems. A blind search strategy for the cube simply would not work. Heuristics programming was invented to solve problems such as this. Using various rules (intelligence), the search space can be drastically reduced so that the problem can be solved in a reasonable amount of time. This is exactly what happens in the Rubik's cube program. As with the basic knowledge representation, the representation of the rules and how they are applied also figure heavily in the program design.

Through this example we will see both the tremendous power and expressiveness of Prolog as well as the obfuscation it sometimes brings as well.

12.1 The Problem

Rubik's cube is a simple looking puzzle. It is a cube with nine tiles on each face. In its solved state each of the sides is made up of tiles of the same color, with a different color for each side. Each of the tiles is actually part of a small cube, or cubie. Each face of the cube (made up of nine cubies) can be rotated. The mechanical genius of the puzzle is that the same cubie can be rotated from multiple sides. A corner cubie can move with three sides, an edge cubie with two sides. Figure 12.1 shows a cube in the initial solved state, and after the right side was rotated 90 degrees clockwise.

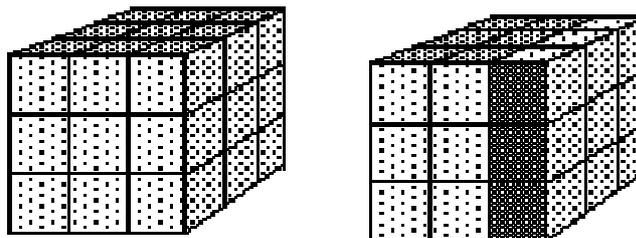


Figure 12.1 A Rubik's Cube before and after the right side is rotated

The problem is to take a cube whose sides have been randomly rotated and figure out how to get it back to the initial solved state. The scrambled cube might look like that of figure 12.2.

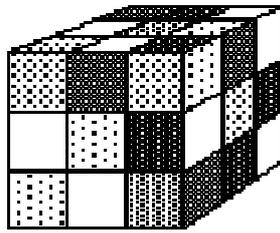


Figure 12.2 A scrambled Rubik's Cube

There are an astronomical number of possible ways to try to unscramble the cube, not very many of which lead to the solved state. To reach a solution using a blind search algorithm is not feasible, even on the largest machines. A human expert can unscramble the cube in well less than a minute.

The difficulty with solving the cube revolves around the fact that if you move one cubie, you have to move seven other cubies as well (the center one doesn't really go anywhere). This is not a big problem in the early stages of unscrambling the cube, but once a number of tiles are positioned correctly, new rotations tend to destroy the solved parts of the cube.

The experienced cube solver knows which complex sequences of moves can be used to manipulate a small portion of the cube without disturbing the other portions of the cube. For example, a 14 move sequence can be used to twist two corner pieces without disturbing any other pieces.

It is important to realize there are actually two different senses of solving the cube. One assumes the problem solver has no previous knowledge of the cube. The other assumes the individual is an expert familiar with all of the intricacies of the cube.

In the first case, the person solving the cube must be able to discover the need for complex sequences of moves and then discover the actual sequences. The program does not have anywhere near the level of "intelligence" necessary to solve the cube in this sense.

In the second case the person is armed with full knowledge of many complex sequences of moves, which can be brought to bear on rearranging various parts of the cube. The problem here is to be able to quickly determine which sequences to apply given a particular scrambled cube. This is the type of "expertise" that is contained in the Rubik's cube program.

In the following sections we will look at how the cube is represented, what is done by searching, what is done with heuristics, how the heuristics are coded, how the cube is manipulated, and how it is displayed.

12.2 The Cube

The core of the program has to be the knowledge representation of the cube and its fundamental rotations.

The cube lends itself to two obvious representation strategies. It can either be viewed simply as 54 tiles, or as 20 cubies (or pieces) each with either two or three tiles. Since much of the intelligence in the program is based on locating pieces and their positions on the cube, a representation that preserves the piece identity is preferred. However, there are

also brute force search predicates, which need a representation that can be manipulated fast. For these predicates a simple flat structure of tiles is best.

The next decision is whether to use flat Prolog data structures (terms) — with each tile represented as an argument of the term, or lists — with each element a tile. Lists are much better for any predicates that might want to search for specific pieces, but they are slower to manipulate as a single entity. Data structures are more difficult to tear apart argument by argument, but are much more efficient to handle as a whole.

(The above statements are true for most Prologs that implement terms using fixed length records. Some Prologs however use lists internally, thus changing the performance trade-offs mentioned above.)

Based on the conflicting design constraints of speed and accessibility, the program actually uses two different notations. One is designed for speed using flat data structures and tiles; the other is a list of cubies, designed for use by the analysis predicates.

The cube is then represented by either the structure:

```
cube(X1, X2, X3, X4, ..., X53, X54)
```

where each **X** represents a tile, or by the list:

```
[p(X1), p(X2), ..., p(X7, X8, X9), ..., p(X31, X32), p(X33, X34),  
...]
```

where each **p(.)** represents a piece. A piece might have one, two, or three arguments, depending on whether it is a center piece, edge piece, or corner piece.

The tiles are each represented by an uppercase letter representing the side of the cube the tile should reside on. These are front, back, top, bottom, right, and left. (The display routine maps the sides to colors.) Quotes are used to indicate the tiles are constants, not variables. Using the constants, the solved state (or goal state of the program) is stored as the Prolog fact **goalstate/1**:

```
goalstate(cube('F', 'R', 'U', 'B', ...)).
```

The ordering of the tiles is not important as long as it is used consistently. The particular ordering chosen starts with the center tiles and then works systematically through the various cubies.

Having decided on two representations, it is necessary to quickly change from one to the other. Unification has exactly the power we need to easily transform between one notation of the cube and the other. A predicate **pieces** takes the flat structure and converts it to a list, or visa versa.

```
pieces(cube(X1, X2, ..., X54), [p(X1), ..., p(X7, X8, X9), ...]).
```

If **Z** is a variable containing a cube in structure notation, then the query

```
?- pieces(Z, Y).
```

will bind the variable **Y** to the same cube in list notation. It can also be used the other way.

The following query can be used to get the goal state in list notation in the variable **PieceState**:

```

?- goalstate(FlatState), pieces(FlatState, PieceState).
FlatState = cube('F', 'R', 'U', 'B', ...).
PieceState = [p('F'), p('R'), ...p('R', 'U'), ...p('B', 'R',
'F'), ...].

```

The first goal unifies **FlatState** with the initial cube we saw earlier. **pieces/2** is then used to generate **PieceState** from **FlatState**.

12.3 Rotation

Unification also gives us the most efficient way to rotate a cube. Each rotation is represented by a predicate, which maps one arrangement of tiles to another. The first argument is the name of the rotation, while the second and third arguments represent a clockwise turn of the side. For example, the rotation of the upper side is represented by:

```

mov(u, cube(X1, ...X6, X7, X8, X9, ...),
cube(X1, ...X6, X20, X19, X21, ...))

```

We can apply this rotation to the top of the goal cube:

```

?- goalstate(State), mov(u, State, NewState).

```

The variable **NewState** would now have a solved cube with the upper side rotated clockwise.

Since these can be used in either direction, we can write a higher level predicate that will make either type of move based on a sign attached to the move.

```

move(+M, OldState, NewState):-
    mov(M, OldState, NewState).
move(-M, OldState, NewState):-
    mov(M, NewState, OldState).

```

Having now built the basic rotations, it is necessary to represent the complex sequences of moves necessary to unscramble the cube. In this case the list notation is the best way to go. For example, a sequence that rotates three corner pieces is represented by:

```

seq(tc3, [+r, -u, -l, +u, -r, -u, +l, +u]).

```

The sequence can be applied to a cube using a recursive list predicate, **move_list/3**:

```

move_list([], X, X).
move_list([Move|T], X, Z):-
    move(Move, X, Y),
    move_list(T, Y, Z).

```

At this point we have a very efficient representation of the cube and a means of rotating it. We next need to apply some expertise to the search for a solution.

12.4 High Level Rules

The most obvious rule for solving Rubik's cube is to attack it one piece at a time. The placing of pieces in the solved cube is done in stages. In Black & Taylor's book they recognize six different stages, which build the cube up from the left side to the right. Some examples of stages are: put the left side edge pieces in place, and put the right side corner pieces in place.

Each stage has from one to four pieces that need placement. One of the advantages of writing expert systems directly in a programming language such as Prolog, is that it is possible to structure the heuristics in an efficient, customized fashion. That is what is done in this program.

The particular knowledge necessary to solve each stage is stored in predicates, which are then used by another predicate, **stage/1**, to set up and solve each stage. Each stage has a plan of pieces it tries to solve for. These are stored in the predicate **pln/2**. It contains the stage number and a list of pieces. For example, stage 5 looks for the four edge pieces on the right side:

```
pln(5, [p('R', 'U'), p('F', 'R'), p('R', 'D'), p('B', 'R')]).
```

Each stage will also use a search routine, which tries various combinations of rotations to position a particular target piece. Different rotations are useful for different stages and, these too, are stored in predicates similar to **pln/2**. The predicate is **cnd/2**, which contains the candidate rotations for the stage.

For example, the first stage (left edge pieces) can be solved using just the simple rotations of the right, upper, and front faces. The last stage (right corner pieces) requires the use of powerful sequences that exchange and twist corner pieces without disturbing the rest of the cube. These have names such as corner-twister 3 and tri-corner 1. They are selected from Black and Taylor's book. These two examples are represented:

```
cnd(1, [r, u, f]).  
cnd(6, [u, tc1, tc3, ct1, ct3]).
```

The **stage/1** predicate drives each of the stages. It basically initializes the stage, and then calls a general purpose routine to improve the cube's state. The initialization of the stage includes setting up the data structures that hold the plan for the stage and the candidate moves. Stage also reorients the cube for the stage to take advantage of symmetries and/or make for better displays.

12.5 Improving the State

The **stage/1** predicate calls the main working predicates, which search for the rotations to put a given piece in place and update all of the appropriate data structures. The representation of the partially solved cube is a key design issue for this portion of the program.

There are predicates in the program that search through a cube in list-piece notation (rather than tile notation) and determine where a piece is or, conversely, which piece is in a given position. These predicates are useful for many portions of the program but are too slow to be used for testing whether a given search has been successful or not. This is true since they not only have to check for the new piece being placed but also have to insure that none of the previously placed pieces have moved.

Unification is again the answer. So far, there are two **cube/54** terms used in the program. One represents the final solved state of the cube, the other represents the current state of the cube. We introduce a third **cube/54**, referred to as the criteria, which is used to denote those tiles which are currently in place and the tiles of the cube that are currently being positioned.

Initially all of the arguments of this third cube are variables. This structure will unify with any cube. As pieces are put in place, the variables representing tiles of the criteria cube are unified with the corresponding tiles of the solved cube. In this case, the criteria cube will only unify with a cube that has those corresponding tiles in place.

As the program attempts to place each piece, it binds another piece in the criteria. For example, as the program attempts to position the sixth piece, the **improve/2** predicate first binds the sixth piece in the criteria with the solved state. At this point then, the first six pieces will have bound values the same as the solved state. The remaining tiles will be represented by unbound variables which unify with anything. The criteria cube will then successfully unify with any cube that has the first six pieces in place.

12.6 The Search

Now that we have a plan of attack on the cube, and a means of representing the current state, and the criteria for testing if a given piece is in place, we can institute a very fast search routine. The core routine to the Rubik's cube program is a predicate **rotate/3**. It is called:

```
rotate(Moves, State, Crit).
```

The variable **Moves** is unbound at calling, and contains the list of moves necessary to position the piece after the search has succeeded. **State** is the current state of the cube, and **Crit** is the criteria for this stage of the solution. **Crit** has all of the pieces found so far bound, as well as the one additional piece for this search. **rotate/3** searches for a sequence of moves which will put the new piece in place without disturbing the existing pieces.

The **rotate/3** predicate illustrates the tremendous power and compactness of Prolog code. At the same time it illustrates the difficulty of understanding some Prolog code. Prolog's power derives from the built in backtracking execution and unification. Both of these features help to eliminate many of the standard programming structures normally used. Thus, a predicate like **rotate/3** has a fraction of the code it would take in another language (and executes fast as well), but it requires a good understanding of the underlying execution behavior of Prolog to understand it.

```
rotate([], State, State).
rotate(Moves, State, Crit):-
    rotate(PriorMoves, State, NextState),
    get_move(ThisMove, NextState, Crit),
    append(PriorMoves, [ThisMove], Moves).
```

The **rotate/3** predicate does a breadth first search as can be seen by the fact that it calls itself recursively before it calls the move generation predicate **get_move/3**. Since the application of moves and testing is so fast, and the depth of search is never great, intermediate results are not saved as in a normal breadth first search. Instead, they are just recalculated each time.

The **append/3** predicate can be used to build lists. In this case it takes **ThisMove** and appends it to the end of the list **PriorMoves**, generating a new list, **Moves**.

The candidate moves for a given stage are stored in a predicate **cand/1** (the actual program is a little more complex) which is maintained by the **stage/1** predicate. For stage one, it would look like:

```
cand(r) .
cand(u) .
cand(f) .
```

The **get_move/3** predicate is called with **Move** unbound, and the second and third arguments bound to the current state and criteria respectively. If the call to **move/3** fails (because it does not rotate the cube into a position which unifies with the criteria), then **cand/1** backtracks generating another possible move. When all of the positive moves fail, then **get_move/3** tries again with negative moves.

```

get_move(+Move, State, Crit):-
    cand(Move),
    mov(Move, State, Crit).
get_move(-Move, State, Crit):-
    cand(Move),
    mov(Move, Crit, State).

```

The efficiencies in **rotate/3** show the rationale behind the early design decisions of cube representation. The **get_move/3** predicate is called with **State** and **Crit**. If it generates a move that unifies with **Crit**, it succeeds, otherwise it fails and backtracks. All of this testing and analysis is done automatically by Prolog's pattern matching call mechanism (unification).

The entire logic of the breadth first search also happens automatically due to the backtracking behavior of Prolog. If **get_move** fails to find a move that reaches the criteria, **rotate/3** backtracks into the recursive call to **rotate/3**. Since the recursive call to **rotate/3** uses **NextState** as the criteria, and **NextState** is unbound, the recursive call will succeed in generating **PriorMoves** and a modified state. Now **get_move/3** tries again with this new state to see if a single move will reach the criteria. This process repeats through as many levels of depth as is necessary to find a sequence of moves that reach the criteria.

In practice, any more than a three-deep search begins to get tedious. The design of the program is such that it does not require more than a three-deep search to find and position any given piece.

12.7 More Heuristics

The program, as described so far, almost works. However, it turns out there are a few situations that will cause the search routines to dig too deep for a solution. These situations drastically affect the performance.

It was necessary to add more intelligence to the program to recognize situations that will not be easily unscrambled by the search routine, and to correct them before calling **rotate/3**.

One of the problems occurs when positioning pieces on the left side. If the piece to be positioned is currently on the right side, then a few simple moves will put it in place on the left side. However, if the piece is already on the left side, but in the wrong position, then it will have to be moved to the right and back to the left. This longer sequence of moves takes longer to search for, so one of the extra heuristics looks for this situation.

The heuristics analyze the cube, test for this condition, and blindly move the piece to the right if it occurs. Then the normal search routine gets it back into its proper place. There are a couple of situations like this that are covered by the heuristics.

It is tempting to think of adding more and more of these heuristics to straighten out the cube with less searching. There is a trade-off, however, and that is it takes time to apply the heuristics, and the search routine is relatively fast. So a heuristic is only worthwhile when the search is slow. The program may be improved by additional heuristics, but the search will still be the core of the program.

12.8 User Interface

A graphical representation of the cube is used to display the progress of the program. A window is kept for recording all of the moves used so far.

In addition the program contains a cube editor that allows you to describe the scrambled cube that has been on your shelf all these years. Just carefully apply the moves step by step and you will get it back to its original state.

12.9 On the Limits of Machines

I don't mind saying that I was pretty proud of myself for writing this program. It was one of my better hacks. At the time, I had a neighbor who was 12 years old and who had just gotten a computer. He loved it and used to come over to my house to hang out with someone who actually got paid for playing with these things.

I had finished an early version of the cube program and decided to knock his socks off. I said, look at this, and ran the program. On my PC-XT it solved a randomly scrambled cube in about three minutes. I looked at him and waited for his awed response — there was nothing. I asked him what he thought. He said he wasn't impressed, his best time was 45 seconds!

Exercises

- 12.1 Improve the speed of the program by experimenting with more heuristics and more canned move sequences. Try to find the optimal balance between the powerful heuristics and sequences and the time it takes to search for them.
- 12.2 Experiment with a version of the cube program, which when given the goal of replacing two pieces without disturbing the others, can "discover" a sequence and remember it for future use.

Appendices - Full Source Code

A Native

Birds Knowledgebase (birds.nkb)

```
% BIRDS.NKB - a sample bird identification system for use with the
% Native shell.

% top_goal where Native starts the inference.

top_goal(X) :-
    bird(X).

order(tubenose) :-
    nostrils(external_tubular),
    live(at_sea),
    bill(hooked).
order(waterfowl) :-
    feet(webbed),
    bill(flat).
order(falconiforms) :-
    eats(meat),
    feet(curved_talons),
    bill(sharp_hooked).
order(passerformes) :-
    feet(one_long_backward_toe).

family(albatross) :-
    order(tubenose),
    size(large),
    wings(long_narrow).
family(swan) :-
    order(waterfowl),
    neck(long),
    color(white),
    flight(ponderous).
family(goose) :-
    order(waterfowl),
    size(plump),
    flight(powerful).
family(duck) :-
    order(waterfowl),
    feed(on_water_surface),
    flight(agile).
family(vulture) :-
    order(falconiforms),
    feed(scavange),
    wings(broad).
family(falcon) :-
    order(falconiforms),
    wings(long_pointed),
    head(large),
    tail(narrow_at_tip).
family(flycatcher) :-
    order(passerformes),
    bill(flat),
    eats(flying_insects).
family(swallow) :-
    order(passerformes),
    wings(long_pointed),
    tail(forked),
    bill(short).
```

```

bird(laysan_albatross) :-
    family(albatross),
    color(white).
bird(black_footed_albatross) :-
    family(albatross),
    color(dark).
bird(fulmar) :-
    order(tubenose),
    size(medium),
    flight(flap_glide).
bird(whistling_swan) :-
    family(swan),
    voice(muffled_musical_whistle).
bird(trumpeter_swan) :-
    family(swan),
    voice(loud_trumpeting).
bird(canada_goose) :-
    family(goose),
    season(winter),           % rules can be further broken down
    country( united_states), % to include regions and migration
    head(black),             % patterns
    cheek(white).
bird(canada_goose) :-
    family(goose),
    season(summer),
    country(canada),
    head(black),
    cheek(white).
bird(snow_goose) :-
    family(goose),
    color(white).
bird(mallard) :-
    family(duck),           % different rules for male
    voice(quack),
    head(green).
bird(mallard) :-
    family(duck),           % and female
    voice(quack),
    color(mottled_brown).
bird(pintail) :-
    family(duck),
    voice(short_whistle).
bird(turkey_vulture) :-
    family(vulture),
    flight_profile(v_shaped).
bird(california_condor) :-
    family(vulture),
    flight_profile(flat).
bird(sparrow_hawk) :-
    family(falcon),
    eats(insects).
bird(peregrine_falcon) :-
    family(falcon),
    eats(birds).
bird(great_crested_flycatcher) :-
    family(flycatcher),
    tail(long_rusty).
bird(ash_throated_flycatcher) :-
    family(flycatcher),
    throat(white).
bird(barn_swallow) :-
    family(swallow),
    tail(forked).

```

```

bird(cliff_swallow) :-
    family(swallow),
    tail(square).
bird(purple_martin) :-
    family(swallow),
    color(dark).

country(united_states) :-
    region(new_england).
country(united_states) :-
    region(south_east).
country(united_states) :-
    region(mid_west).
country(united_states) :-
    region(south_west).
country(united_states) :-
    region(north_west).
country(united_states) :-
    region(mid_atlantic).
country(canada) :-
    province(ontario).
country(canada) :-
    province(quebec).
country(canada) :-
    province(etc).

region(new_england) :-
    state(X),
    member(X, [massachusetts, vermont, etc]).
region(south_east) :-
    state(X),
    member(X, [florida, mississippi, etc]).
region(canada) :-
    province(X),
    member(X, [ontario, quebec, etc]).

nostrils(X) :-
    ask(nostrils, X).

live(X) :-
    ask(live, X).

bill(X) :-
    ask(bill, X).

size(X) :-
    menuask(size, X, [large, plump, medium, small]).

eats(X) :-
    ask(eats, X).

feet(X) :-
    ask(feet, X).

wings(X) :-
    ask(wings, X).

neck(X) :-

```

```
ask(neck,X).

color(X) :-
    ask(color,X).

flight(X) :-
    menuask(flight,X,[ponderous,powerful,agile,flap_glide,other]).

feed(X) :-
    ask(feed,X).

head(X) :-
    ask(head,X).

tail(X) :-
    menuask(tail,X,[narrow_at_tip,forked,long_rusty,square,other]).

voice(X) :-
    ask(voice,X).

season(X) :-
    menuask(season,X,[winter,summer]).

cheek(X) :-
    ask(cheek,X).

flight_profile(X) :-
    menuask(flight_profile,X,[flat,v_shaped,other]).

throat(X) :-
    ask(throat,X).

state(X) :-
    menuask(state,X,[massachusetts,vermont,florida,mississippi,etc]).

province(X) :-
    menuask(province,X,[ontario,quebec,etc]).

multivalued(voice).
multivalued(color).
multivalued(eats).
```

Native Shell (native.pro)

```
% Native - a simple shell for use with Prolog
% knowledge bases. It includes explanations.

:-op(900,xfy, :).

main :-
    greeting,
    repeat,
    write('> '),
    read(X),
    do(X),
    X == quit.

greeting :-
    write('This is the native Prolog shell.'), nl,
    native_help.

do(help) :-
    native_help,
    !.
do(load) :-
    load_kb,
    !.
do(solve) :-
    solve,
    !.
do(how(Goal)) :-
    how(Goal),
    !.
do(whynot(Goal)) :-
    whynot(Goal),
    !.
do(quit).
do(X) :-
    write(X),
    write(' is not a legal command.'), nl,
    fail.

native_help :-
    write('Type help. load. solve. how(Goal). whynot(Goal). or quit.'), nl,
    write('at the prompt.'), nl.

load_kb :-
    write('Enter file name in single quotes (ex. ''birds.nkb'').: '),
    read(F),
    reconsult(F).

solve :-
    abolish(known,3),
    prove(top_goal(X),[]),
    write('The answer is '), write(X), nl.
solve :-
    write('No answer found.'), nl.

ask(Attribute,Value,_) :-
    known(yes,Attribute,Value), % succeed if we know its true
    !. % and dont look any further
ask(Attribute,Value,_) :-
```

```

    known(_,Attribute,Value),      % fail if we know its false
    !,
    fail.
ask(Attribute,_,_) :-
    not multivalued(Attribute),
    known(yes,Attribute,_),        % fail if its some other value.
    !,                             % this ensures this is the wrong value
    fail.
ask(A,V,Hist) :-
    write(A :V),                  % if we get here, we need to ask.
    write('? (yes or no) '),
    get_user(Y,Hist),             % get the answer
    asserta(known(Y,A,V)),        % remember it so we dont ask again.
    Y = yes.                      % succeed or fail based on answer.

% "menuask" is like ask, only it gives the user a menu to to choose
% from rather than a yes on no answer. In this case there is no
% need to check for a negative since "menuask" ensures there will
% be some positive answer.

menuask(Attribute,Value,_,_) :-
    known(yes,Attribute,Value), % succeed if we know
    !.
menuask(Attribute,_,_,_) :-
    known(yes,Attribute,_),      % fail if its some other value
    !, fail.
menuask(Attribute,AskValue,Menu,Hist) :-
    nl,
    write('What is the value for '), write(Attribute), write('?'), nl,
    display_menu(Menu),
    write('Enter the number of choice> '),
    get_user(Num,Hist),nl,
    pick_menu(Num,AnswerValue,Menu),
    asserta(known(yes,Attribute,AnswerValue)),
    AskValue = AnswerValue. % succeed or fail based on answer

display_menu(Menu) :-
    disp_menu(1,Menu),
    !. % make sure we fail on backtracking

disp_menu(_, []).
disp_menu(N, [Item|Rest]) :-
% recursively write the head of the list and disp_menu the tail
    write(N), write(' : '), write(Item), nl,
    NN is N + 1,
    disp_menu(NN,Rest).

pick_menu(N,Val,Menu) :-
    integer(N),                  % make sure they gave a number
    pic_menu(1,N,Val,Menu),      % start at one
    !.
pick_menu(Val,Val,_). % if they didn't enter a number, use
% what they entered as the value
pic_menu(_,_,none_of_the_above, []). % if we've exhausted the list
pic_menu(N,N,Item,[Item|_]). % the counter matches the number
pic_menu(Ctr,N,Val,[_|Rest]) :-
    NextCtr is Ctr + 1,          % try the next one
    pic_menu(NextCtr,N,Val,Rest).

get_user(X,Hist) :-
    repeat,
    write('> '),
    read(X),

```

```

    process_ans(X,Hist),
    !.

process_ans(why,Hist) :-
    write_list(4,Hist),
    !,
    fail.
process_ans(X,_).

% Prolog in Prolog for explanations.
% It is a bit confusing because of the ambiguous use of the comma, both
% to separate arguments and as an infix operator between the goals of
% a clause.

prove(true,_) :-
    !.
prove((Goal,Rest),Hist) :-
    prov(Goal,[Goal|Hist]),
    prove(Rest,Hist).
prove(Goal,Hist) :-
    prov(Goal,[Goal|Hist]).

prov(true,_) :-
    !.
prov(menuask(X,Y,Z),Hist) :-
    menuask(X,Y,Z,Hist),
    !.
prov(ask(X,Y),Hist) :-
    ask(X,Y,Hist),
    !.
prov(Goal,Hist) :-
    clause(Goal,Body),
    prove(Body,Hist).

% EXPLANATIONS

how(Goal) :-
    clause(Goal,Body),
    prove(Body,[]),
    write_body(4,Body).

whynot(Goal) :-
    clause(Goal,Body),
    write_line([Goal,'fails because: ']),
    explain(Body).
whynot(_).

explain(true).
explain((Head,Body)) :-
    check(Head),
    explain(Body).

check(H) :-
    prove(H,[]),
    write_line([H,succeeds]),
    !.
check(H) :-
    write_line([H,fails]),
    fail.

write_list(N,[]).

```

```
write_list(N, [H|T]) :-
    tab(N), write(H), nl,
    write_list(N,T).

write_body(N, (First,Rest)) :-
    tab(N), write(First), nl,
    write_body(N,Rest).
write_body(N,Last) :-
    tab(N), write(Last), nl.

write_line(L) :-
    flatten(L,LF),
    write_lin(LF).

write_lin([]) :-
    nl.
write_lin([H|T]) :-
    write(H), tab(1),
    write_lin(T).

flatten([],[]) :-
    !.
flatten([[ ]|T],T2) :-
    flatten(T,T2),
    !.
flatten([[X|Y]|T], L) :-
    flatten([X|[Y|T]],L),
    !.
flatten([H|T], [H|T2]) :-
    flatten(T,T2).
```

B Clam

Car Knowledgebase (car.ckb)

```
goal problem.

rule 1
  if not turn_over and
    battery_bad
  then problem is battery cf 100.

rule 2
  if lights_weak
  then battery_bad cf 50.

rule 3
  if radio_weak
  then battery_bad cf 50.

rule 4
  if turn_over and
    smell_gas
  then problem is flooded cf 80.

rule 5
  if turn_over and
    gas_gauge is empty
  then problem is out_of_gas cf 90.

rule 6
  if turn_over and
    gas_gauge is low
  then problem is out_of_gas cf 30.

output problem is battery get the battery recharged.
output problem is out_of_gas start walking or hitching to a gas station.
output problem is flooded wait 5 minutes and try again.

ask turn_over
  menu (yes no)
  prompt 'Does the engine turn over?'.

ask lights_weak
  menu (yes no)
  prompt 'Are the lights weak?'.

ask radio_weak
  menu (yes no)
  prompt 'Is the radio weak?'.

ask smell_gas
  menu (yes no)
  prompt 'Do you smell gas?'.

ask gas_gauge
  menu (empty low full)
  prompt 'What does the gas gauge say?'.

```

Birds Knowledgebase (birds.ckb)

```
goal bird.

rule 1
  if  nostrils is external_tubular and
      live is at_sea and
      bill is hooked
  then order is tubenose cf 80.

rule 2
  if  feet is webbed and
      bill is flat
  then order is waterfowl cf 80.

rule 3
  if  eats is meat and
      feet is curved_talons and
      bill is sharp_hooked
  then order is falconiforms cf 80.

rule 4
  if  feet is one_long_backward_toe
  then order is passerformes cf 80.

rule 5
  if  order is tubenose and
      size is large and
      wings is long_narrow
  then family is albatross cf 80.

rule 6
  if  order is waterfowl and
      neck is long and
      color is white and
      flight is ponderous
  then family is swan cf 80.

rule 7
  if  order is waterfowl and
      size is plump and
      flight is powerful
  then family is goose cf 80.

rule 8
  if  order is waterfowl and
      feed is on_water_surface and
      flight is agile
  then family is duck cf 80.

rule 9
  if  order is falconiforms and
      feed is scavage and
      wings is broad
  then family is vulture cf 80.

rule 10
  if  order is falconiforms and
      wings is long_pointed and
      head is large and
      tail is narrow_at_tip
  then family is falcon cf 80.

rule 11
  if  order is passerformes and
      bill is flat and
      eats is flying_insects
```

```
    then family is flycatcher cf 80.

rule 12
  if  order is passerformes and
     wings is long_pointed and
     tail is forked and
     bill is short
  then family is swallow cf 80.

rule 13
  if  family is albatross and
     color is white
  then bird is laysan_albatross cf 80.

rule 14
  if  family is albatross and
     color is dark
  then bird is black_footed_albatross cf 80.

rule 15
  if  order is tubenose and
     size is medium and
     flight is flap_glide
  then bird is fulmar cf 80.

rule 16
  if  family is swan and
     voice is muffled_musical_whistle
  then bird is whistling_swan cf 80.

rule 17
  if  family is swan and
     voice is loud_trumpeting
  then bird is trumpeter_swan cf 80.

rule 18
  if  family is goose and
     season is winter and
     country is united_states and
     head is black and
     cheek is white
  then bird is canada_goose cf 80.

rule 19
  if  family is goose and
     season is summer and
     country is canada and
     head is black and
     cheek is white
  then bird is canada_goose cf 80.

rule 20
  if  family is goose and
     color is white
  then bird is snow_goose cf 80.

rule 21
  if  family is duck and
     voice is quack and
     head is green
  then bird is mallard cf 80.

rule 22
  if  family is duck and
     voice is quack and
     color is mottled_brown
  then bird is mallard cf 80.
```

```
rule 23
  if family is duck and
     voice is short_whistle
  then bird is pintail cf 80.

rule 24
  if family is vulture and
     flight_profile is v_shaped
  then bird is turkey_vulture cf 80.

rule 25
  if family is vulture and
     flight_profile is flat
  then bird is california_condor cf 80.

rule 26
  if family is falcon and
     eats is insects
  then bird is sparrow_hawk cf 80.

rule 27
  if family is falcon and
     eats is birds
  then bird is peregrine_falcon cf 80.

rule 28
  if family is flycatcher and
     tail is long_rusty
  then bird is great_crested_flycatcher cf 80.

rule 29
  if family is flycatcher and
     throat is white
  then bird is ash_throated_flycatcher cf 80.

rule 30
  if family is swallow and
     tail is forked
  then bird is barn_swallow cf 80.

rule 31
  if family is swallow and
     tail is square
  then bird is cliff_swallow cf 80.

rule 32
  if family is swallow and
     color is dark
  then bird is purple_martin cf 80.

rule 33
  if region is new_england
  then country is united_states.

rule 34
  if region is south_east
  then country is united_states.

rule 35
  if region is mid_west
  then country is united_states.

rule 36
  if region is south_west
  then country is united_states.

rule 37
  if region is north_west
```

```

    then country is united_states.

rule 38
    if    region is mid_atlantic
    then country is united_states.

rule 39
    if    region is ontario
    then country is canada.

rule 40
    if    region is quebec
    then country is canada.

ask bill
    menu (hooked flat sharp_hooked short other)
    prompt 'What type of bill?'.

ask cheek
    menu (white other)
    prompt 'What type of cheek?'.

ask color
    menu (white dark mottled_brown other)
    prompt 'What color is it?'.

ask region
    menu (new_england south_east mid_west south_west
         north_west mid_atlantic ontario quebec other)
    prompt 'What region was it seen in?'.

ask eats
    menu (meat flying_insects insects birds other)
    prompt 'What does it eat?'.

ask feed
    menu (on_water_surface scavage other)
    prompt 'Where does it feed?'.

ask feet
    menu (webbed curved_talons one_long_backward_toe other)
    prompt 'What type of feet?'.

ask flight
    menu (ponderous powerful agile flap_glide other)
    prompt 'What type of flight?'.

ask flight_profile
    menu (v_shaped flat other)
    prompt 'What is the flight profile?'.

ask head
    menu (large black green other)
    prompt 'What type of head?'.

ask live
    menu (at_sea other)
    prompt 'Where does it live?'.

ask neck
    menu (long other)
    prompt 'What type of neck does it have?'.

ask nostrils
    menu (external_tubular other)
    prompt 'What type of nostrils?'.

ask season

```

```
    menu (summer fall winter spring)
    prompt 'What season was it seen in?'.

ask size
    menu (large medium small plump other)
    prompt 'What size is it?'.

ask tail
    menu (narrow_at_tip forked long_rusty square other)
    prompt 'What type of tail?'.

ask throat
    menu (white other)
    prompt 'What type of throat?'.

ask voice
    menu (muffled_musical_whistle loud_trumpeting quack short_whistle other)
    prompt 'What type of voice?'.

ask wings
    menu (long_narrow broad long_pointed other)
    prompt 'What type of wings does it have?'.

multivalued voice.

multivalued color.

multivalued eats.
```

Clam Shell (clam.pro)

```
% Clam - expert system shell with EMYCIN type certainty factors

% This system is an imitation of the EMYCIN imitators. It does backward
% chaining (goal directed) inference with uncertainty. The uncertainty
% is modelled using the MYCIN certainty factors.

% The only data structure is an attribute:value pair.

% NOTE - CF calculation in update only good for positive CF

main :-
    do_over,
    super.

% The main command loop

super :-
    repeat,
        write('consult restart load list trace on/off how exit'), nl,
        write('> '),
        read_line([X|Y]),
        doit([X|Y]),
    X == exit.

doit([consult]) :-
    top_goals,
    !.
doit([restart]) :-
    do_over,
    !.
doit([load]) :-
    load_rules,
    !.
doit([list]) :-
    list_facts,
    !.
doit([trace,X]) :-
    set_trace(X),
    !.
doit([how|Y]) :-
    how(Y),
    !.
doit([exit]).
doit([X|Y]) :-
    write('invalid command : '),
    write([X|Y]), nl.

% top_goals works through each of the goals in sequence

top_goals :-
    ghoul(Attr),
    top(Attr),
    print_goal(Attr),
    fail.
top_goals.

% top starts the backward chaining by looking for rules that reference
% the attribute in the RHS. If it is known with certainty 100, then
% no other rules are tried, and other candidates are eliminated. Otherwise
% other rules which might yield different values for the attribute
```

```

% are tried as well

top(Attr) :-
    findgoal(av(Attr,Val),CF,[goal(Attr)]),
    !.
top(_) :-
    true.

% prints all hypotheses for a given attribute

print_goal(Attr) :-
    nl,
    fact(av(Attr,X),CF,_),
    CF >= 20,
    outp(av(Attr,X),CF), nl,
    fail.
print_goal(Attr) :-
    write('done with '), write(Attr), nl,
    nl.

outp(av(A,V),CF) :-
    output(A,V,PrintList),
    pretty(av(A,V), X),
    printlist(X),
    tab(1), write(cf(CF)), write(': '),
    printlist(PrintList),
    !.
outp(av(A,V),CF) :-
    pretty(av(A,V), X),
    printlist(X),
    tab(1), write(cf(CF)).

printlist([]).
printlist([H|T]) :-
    write(H), tab(1),
    printlist(T).

% findgoal is the guts of the inference. It copes with already known
% attribute value pairs, multivalued attributes and single valued
% attributes. It uses the EMYCIN certainty factor arithmetic to
% propagate uncertainties.

% 1 - if its recorded and the value matches, we're done, if the
%     value doesn't match, but its single valued and known with
%     certainty 100 definitely fail

findgoal(X,Y,_):-
    bugdisp([' ',X]),
    fail.
findgoal(not Goal,NCF,Hist) :-
    findgoal(Goal,CF,Hist),
    NCF is - CF,
    !.
findgoal(Goal,CF,Hist) :-
    fact(Goal,CF,_),
    !.
%findgoal(av(Attr,Val),CF) :-
% bound(Val),
% fact(av(Attr,V,_),CF),
% Val \= V,
% single_valued(Attr),
% CF=100,
% !,
% fail.

```

```

% 2 - if its askable, just ask and record the answer

findgoal(Goal,CF,Hist) :-
    can_ask(Goal,Hist),
    !,
    findgoal(Goal,CF,Hist).

% 3 - find a rule with the required attribute on the RHS. try to prove
% the LHS. If its proved, use the certainty of the LHS combined
% with the certainty of the RHS to compute the cf of the derived
% result

findgoal(Goal,CurCF,Hist) :-
    fg(Goal,CurCF,Hist).

fg(Goal,CurCF,Hist) :-
    rule(N,lhs(IfList), rhs(Goal,CF)),
    bugdisp(['call rule',N]),
    prove(N,IfList,Tally,Hist),
    bugdisp(['exit rule',N]),
    adjust(CF,Tally,NewCF),
    update(Goal,NewCF,CurCF,N),
    CurCF == 100,
    !.
fg(Goal,CF,_) :-
    fact(Goal,CF,_).

% can_ask shows how to query the user for various types of goal patterns

can_ask(av(Attr,Val),Hist) :-
    not asked(av(Attr,_)),
    askable(Attr,Menu>Edit,Prompt),
    query_user(Attr,Prompt,Menu>Edit,Hist),
    asserta( asked(av(Attr,_)) ).

% answer the how question at the top level, to explain how an answer was
% derived. It can be called successive times to get the whole proof.

how([]) :-
    write('Goal? '), read_line(X), nl,
    pretty(Goal,X),
    how(Goal).
how(X) :-
    pretty(Goal,X),
    nl,
    how(Goal).
how(not Goal) :-
    fact(Goal,CF,Rules),
    CF < -20,
    pretty(not Goal,PG),
    write_line([PG,was,derived,from,'rules: '|Rules]),
    nl,
    list_rules(Rules),
    fail.
how(Goal) :-
    fact(Goal,CF,Rules),
    CF > 20,
    pretty(Goal,PG),
    write_line([PG,was,derived,from,'rules: '|Rules]),
    nl,
    list_rules(Rules),
    fail.
how(_).

```

```

list_rules([]).
list_rules([R|X]) :-
    list_rule(R),
    % how_lhs(R),
    list_rules(X).

list_rule(N) :-
    rule(N, lhs(Iflist), rhs(Goal,CF)),
    write_line(['rule ',N]),
    write_line([' If']),
    write_ifs(Iflist),
    write_line([' Then']),
    pretty(Goal,PG),
    write_line([' ',PG,CF]), nl.

write_ifs([]).
write_ifs([H|T]) :-
    pretty(H,HP),
    tab(4), write_line(HP),
    write_ifs(T).

pretty(av(A,yes),[A]) :-
    !.
pretty(not av(A,yes), [not,A]) :-
    !.
pretty(av(A,no),[not,A]) :-
    !.
pretty(not av(A,V),[not,A,is,V]).
pretty(av(A,V),[A,is,V]).

how_lhs(N) :-
    rule(N,lhs(Iflist),_),
    !,
    how_ifs(Iflist).

how_ifs([]).
how_ifs([Goal|X]) :-
    how(Goal),
    how_ifs(X).

% get input from the user. Either a straight answer from the menu, or
% an answer with cf N appended to it.

query_user(Attr,Prompt,[yes,no],_,Hist) :-
    !,
    write(Prompt), nl,
    get_user(X,Hist),
    get_vcf(X,Val,CF),
    asserta(fact(av(Attr,Val),CF,[user])).
query_user(Attr,Prompt,Menu,Edit,Hist) :-
    write(Prompt), nl,
    menu_read(VList,Menu,Hist),
    assert_list(Attr,VList).

menu_read(X,Menu,Hist) :-
    write_list(2,Menu),
    get_user(X,Hist).

get_user(X,Hist) :-
    repeat,
    write(': '),

```

```

    read_line(X),
    process_ans(X,Hist).

process_ans([why],Hist) :-
    nl, write_hist(Hist),
    !,
    fail.
process_ans(X,_).

write_hist([]) :-
    nl.
write_hist([goal(X)|T]) :-
    write_line([goal,X]),
    !,
    write_hist(T).
write_hist([N|T]) :-
    list_rule(N),
    !,
    write_hist(T).

write_list(N, []).
write_list(N, [H|T]) :-
    tab(N), write(H), nl,
    write_list(N,T).

assert_list(_, []).
assert_list(Attr, [not,Val,cf,CF|X]) :-
    !,
    NCF is - CF,
    asserta(fact(av(Attr,Val),NCF,[user])),
    assert_list(Attr,X).
assert_list(Attr, [not,Val|X]) :-
    !,
    asserta(fact(av(Attr,Val),-100,[user])),
    assert_list(Attr,X).
assert_list(Attr, [Val,cf,CF|X]) :-
    !,
    asserta(fact(av(Attr,Val),CF,[user])),
    assert_list(Attr,X).
assert_list(Attr, [Val|X]) :-
    asserta(fact(av(Attr,Val),100,[user])),
    assert_list(Attr,X).

get_vcf([no],yes,-100).
get_vcf([no,CF],yes,NCF) :-
    NCF is -CF.
get_vcf([no,cf,CF],yes,NCF) :-
    NCF is -CF.
get_vcf([Val,CF],Val,CF).
get_vcf([Val,cf,CF],Val,CF).
get_vcf([Val],Val,100).
get_vcf([not,Val],Val,-100).
get_vcf([not,Val,CF],Val,NCF) :-
    NCF is -CF.
get_vcf([not,Val,cf,CF],Val,NCF) :-
    NCF is -CF.

% prove works through a LHS list of premises, calling findgoal on
% each one. the total cf is computed as the minimum cf in the list

prove(N,IfList,Tally,Hist) :-
    prov(IfList,100,Tally,[N|Hist]),

```

```

!.
prove(N,_,_) :-
    bugdisp(['fail rule',N]),
    fail.

prov([],Tally,Tally,Hist).
prov([H|T],CurTal,Tally,Hist) :-
    findgoal(H,CF,Hist),
    minimum(CurTal,CF,Tal),
    Tal >= 20,
    prov(T,Tal,Tally,Hist).

% update - if its already known with a given cf, here is the formula
% for adding in the new cf. This is used in those cases where multiple
% RHS reference the same attr :val

update(Goal,NewCF,CF,RuleN) :-
    fact(Goal,OldCF,_),
    combine(NewCF,OldCF,CF),
    retract(fact(Goal,OldCF,OldRules)),
    asserta(fact(Goal,CF,[RuleN|OldRules])),
    (CF == 100, single_valued(Attr), erase_other(Attr)
    ;
    true
    ),
    !.
update(Goal,CF,CF,RuleN) :-
    asserta(fact(Goal,CF,[RuleN])).

erase_other(Attr) :-
    fact(av(Attr,Val),CF,_),
    CF < 100,
    retract(fact(av(Attr,Val),CF,_)),
    fail.
erase_other(Attr) :-
    true.

adjust(CF1,CF2,CF) :-
    X is CF1 * CF2 / 100,
    int_round(X,CF).

combine(CF1,CF2,CF) :-
    CF1 >= 0,
    CF2 >= 0,
    X is CF1 + CF2 * (100 - CF1) / 100,
    int_round(X,CF).
combine(CF1,CF2,CF) :-
    CF1 < 0,
    CF2 < 0,
    X is - (-CF1 -CF2 * (100 + CF1) / 100),
    int_round(X,CF).
combine(CF1,CF2,CF) :-
    (CF1 < 0; CF2 < 0),
    (CF1 > 0; CF2 > 0),
    abs_minimum(CF1,CF2,MCF),
    X is 100 * (CF1 + CF2) / (100 - MCF),
    int_round(X,CF).

abs_minimum(A,B,X) :-
    absolute(A, AA),
    absolute(B, BB),
    minimum(AA,BB,X).

```

```

absolute(X, X) :-
    X >= 0.
absolute(X, Y) :-
    X < 0,
    Y is -X.

%minimum(A,B,A) :-
% A =< B.
%minimum(A,B,B) :-
% B > A.

%min([],X,X).
%min([H|T],Z,X) :-
% H < Z,
% min(T,H,X).
%min([H|T],Z,X) :-
% H >= Z,
% min(T,Z,X).

minimum(X,Y,X) :-
    X =< Y,
    !.
minimum(X,Y,Y) :-
    Y =< X.

int_round(X,I) :-
    X >= 0,
    I is integer(X + 0.5).
int_round(X,I) :-
    X < 0,
    I is integer(X - 0.5).

set_trace(off) :-
    ruletrace,
    retract(ruletrace).
set_trace(on) :-
    not ruletrace,
    asserta(ruletrace).
set_trace(_).

single_valued(A) :-
    multivalued(A),
    !,
    fail.
single_valued(A) :-
    true.

list_facts :-
    fact(X,Y,_),
    write(fact(X,Y)), nl,
    fail.
list_facts :-
    true.

do_over :-
    abolish(asked,1),
    abolish(fact,3).

```

```

clear :-
  abolish(asked,1),
  abolish(fact,3),
  abolish(rule,1),
  abolish(multivalued,1),
  abolish(askable,1),
  abolish(ghoul,1).

blank_lines(0).
blank_lines(N) :-
  nl,
  NN is N - 1,
  blank_lines(NN).

bugdisp(L) :-
  ruletrace,
  write_line(L),
  !.
bugdisp(_).

write_line(L) :-
  flatten(L,LF),
  write_lin(LF).

write_lin([]) :-
  nl.
write_lin([H|T]) :-
  write(H), tab(1),
  write_lin(T).

flatten([],[]) :-
  !.
flatten([[_|T],T2) :-
  flatten(T,T2),
  !.
flatten([[X|Y]|T], L) :-
  flatten([X|[Y|T]],L),
  !.
flatten([H|T],[H|T2]) :-
  flatten(T,T2).

member(X,[X|_]).
member(X,[_|Z]) :-
  member(X,Z).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% LDRULES - this module reads a rule file and translates it to internal
%           Prolog format for the Clam shell
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

load_rules :-
  write('Enter file name in single quotes (ex. 'car.ckb'.): '),
  read(F),
  load_rules(F).

load_rules(F) :-
  clear_db,
  see(F),
  lod_ruls,
  write('rules loaded'), nl,
  seen,

```

```

!.

lod_ruls :-
    repeat,
    read_sentence(L),
%   bug(L),
    process(L),
    L == ['!EOF'].

process(['!EOF']) :-
    !.
process(L) :-
    trans(R,L,[]),
    bug(R),
    assertz(R),
    !.
process(L) :-
    write('trans error on:'), nl,
    write(L), nl.

clear_db :-
    abolish(cf_model,1),
    abolish(ghoul,1),
    abolish(askable,4),
    abolish(output,3),
    abolish(rule,3).

bug(cf_model(X)) :-
    write(cf_model(X)), nl,
    !.
bug(ghoul(X)) :-
    write(ghoul(X)), nl,
    !.
bug(askable(A,_,_,_)) :-
    write('askable '), write(A), nl,
    !.
bug(output(A,V,PL)) :-
    write('output '), write(V), nl,
    !.
bug(rule(N,_,_)) :-
    write('rule '), write(N), nl,
    !.
bug(X) :-
    write(X), nl.

% trans - translates a list of atoms in external rule form to internal
%         rule form

trans(cf_model(X)) -->
    [cf,model,X].
trans(cf_model(X)) -->
    [cf,model,is,X].
trans(cf_model(X)) -->
    [cf,X].
trans(ghoul(X)) -->
    [goal,is,X].
trans(ghoul(X)) -->
    [goal,X].
trans(askable(A,M,E,P)) -->
    [ask,A],
    menux(M),
    editchk(E),
    prompt(A,P).

```

```

trans(output(A,V,PL)) -->
  [output],
  phraz(av(A,V)),
  plist(PL).
trans(rule(N, lhs(IF), rhs(THEN,CF))) -->
  id(N),
  if(IF),
  then(THEN,CF).
trans(multivalued(X)) -->
  [multivalued,X].
trans('Parsing error'-L,L,_).

%default(D) -->
% [default,D].
%default(none) -->
% [].

menux(M) -->
  [menu,'('],
  menuxlist(M).

menuxlist([Item]) -->
  [Item,')'].
menuxlist([Item|T]) -->
  [Item],
  menuxlist(T).

editchk(E) -->
  [edit,E].
editchk(none) -->
  [].

prompt(_,P) -->
  [prompt,P].
prompt(P,P) -->
  [].

id(N) -->
  [rule,N].

if(IF) -->
  [if],
  iflist(IF).

iflist([IF]) -->
  phraz(IF),
  [then].
iflist([Hif|Tif]) -->
  phraz(Hif),
  [and],
  iflist(Tif).
iflist([Hif|Tif]) -->
  phraz(Hif),
  [' ',''],
  iflist(Tif).

then(THEN,CF) -->
  phraz(THEN),
  [cf],

```

```

[CF].
then(THEN,100) -->
  phraz(THEN).

phraz(not av(Attr,yes)) -->
  [not,Attr].
phraz(not av(Attr,yes)) -->
  [not,a,Attr].
phraz(not av(Attr,yes)) -->
  [not,an,Attr].
phraz(not av(Attr,Val)) -->
  [not,Attr,is,Val].
phraz(not av(Attr,Val)) -->
  [not,Attr,are,Val].
phraz(av(Attr,Val)) -->
  [Attr,is,Val].
phraz(av(Attr,Val)) -->
  [Attr,are,Val].
phraz(av(Attr,yes)) -->
  [Attr].

plist([Text]) -->
  [Text].
plist([Htext|Ttext]) -->
  [Htext],
  plist(Ttext).

%%
%% end LDRULS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

read_line(L) :-
  read_word_list([13,10], L),
  !.

read_sentence(S) :-
  read_word_list(['`'], S),
  !.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% From the Cogent Prolog Toolbox
%%
%% rwl.pro - read word list, based on Clocksin & Mellish
%%
%% Read word list reads in a list of chars (terminated with a !, . or ?)
%% and converts it to a list of atomic entries (including numbers).
%% Uppercase is converted to lower case.
%% A 'word' is one item in our generated list

%% This version has been modified for CLAM by allowing an additional
%% argument, Xs, that is a list of the ending characters. This allows the
%% code to be used for both command input, terminated by the Enter key, and
%% reading the knowledge base files, terminated after multiple lines by
%% a period.

%% It has further been modified to skip everything between a % and the
%% end of line, allowing for Prolog style comments.

read_word_list(LW,[W|Ws]) :-
  get0(C),
  readword(C, W, C1),           % Read word starting with C, C1 is first new
  restsent(LW, C1, Ws).        % character - use it to get rest of sentence

restsent(_, '!EOF', []).

```



```

in_word(`-,`-).
in_word(`_,`_).

% Have character C (known integer) - keep reading integers and build
% up the number until we hit a non-integer. Return this in C1, and
% return the computed number in W.

number_word(C, W, C1, Pow10) :-
    is_num(C),
    !,
    get0(C2),
    number_word(C2, W1, C1, P10),
    Pow10 is P10 * 10,
    W is integer(((C - `0) * Pow10) + W1).
number_word(C, 0, C, 0.1).

is_num(C) :-
    C =< `9,
    C >= `0.

% These symbols delineate end of sentence

%lastword(`.).
%lastword(`!).
%lastword(`?).
%lastword(13).    % carriage return
%lastword(10).   % line feed

%%
%% end RWL.PRO from Cogent Prolog Toolbox
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

Build Rules (bldrules.pro)

```
% build rules use dcg in reverse to make clam rules from Prolog rules
% You can use bldrules.pro to convert, for example, the native Prolog
% rules of the birds.pro into clam syntax.

main :-
    write($From file: $), read(From),
    write($To file: $), read(To),
    doit(From,To).

doit(From,To) :-
    see(From),
    tell(To),
    test.

test :-
    cntr_set(1,1),
    repeat,
    read(X),
    tran(X,Ans,[]),
    write_nice(Ans), nl,
    X == '!EOF'.
test :-
    told,
    seen,
    write(done).

xxif(Body) -->
    [if],
    xxbody(Body).

xxthen(Head) -->
    {Head =.. [F,A]},
    [then,F,is,A].

xxbody((H,T)) -->
    {!,H =.. [F,A]},
    [F,is,A,and],
    xxbody(T).
xxbody(H) -->
    {H =.. [F,A]},
    [F,is,A].

tran(A,B,C) :-
    trans(A,B,C),
    !.
tran(X,X,_).

trans('!EOF','!EOF',_).

trans((Head :- true)) -->
    {Head =.. [F,A]},
    [F,is,A],
    !.
trans((Head :- Body)) -->
    {cntr_get(1,ID)},
    [rule, ID],
    xxif(Body),
```

```

xxthen(Head),
{ID2 is ID + 1,
cntr_set(1, ID2)}.

write_nice(X) :-
  wr_nice(X),
  !.

wr_nice([]) :-
  !,
  write('.'), nl.
wr_nice([if|T]) :-
  !,
  nl, write(' if '),
  wr_nice(T).
wr_nice([then|T]) :-
  !,
  nl, write(' then '),
  wr_nice(T).
wr_nice([and|T]) :-
  !,
  write(and), nl, write(' '),
  wr_nice(T).
wr_nice([H|T]) :-
  !,
  write(H), write(' '),
  wr_nice(T).
wr_nice(X) :-
  write(X).

```


C Oops

Room Knowledgebase (room.okb)

```
% ROOM is an expert system for placing furniture in a living room.

% It is written using the OOPS production system rules language.

% It is only designed to illustrate the use of a forward chaining
% rules based language for solving configuration problems. As such
% it makes many simplifying assumptions (such as furniture has no
% width). It just decides which wall each item goes on, and does
% not decide the relative placement on the wall.

% Furniture to be placed in the room is stored in terms of the form
% "furniture(item,length)". The rules look for unplaced furniture,
% and if found attempt to place it according to the rules of thumb.
% Once placed, the available space on a wall is updated, the furniture
% is ed on a wall with a term of the form "position(item,wall)",
% and the original "furniture" term is removed.

% These are the terms which are initially stored in working storage.
% They set a goal used to force firing of certain preliminary rules,
% and various facts about the problem domain used by the actual
% configuration rules.

initial_data([goal(place_furniture),
             not_end_yet,
             legal_furniture([couch, chair, table_lamp, end_table,
                              coffee_table, tv, standing_lamp, end]),
             opposite(north,south),
             opposite(south,north),
             opposite(east,west),
             opposite(west,east),
             right(north,west),
             right(west,south),
             right(south,east),
             right(east,north),
             left(north,east),
             left(east,south),
             left(south,west),
             left(west,north)]).

% Rules 1-8 are an example of how to generate procedural behavior
% from a non-procedural rule language. These rules force a series
% of prompts and gather data from the user on the room and furniture
% to be configured. They are included to illustrate the kludgy
% nature of production systems in a conventional setting.

% This is in contrast to rules f1-f14 which elegantly configure the room.

rule 1:
  [1: goal(place_furniture),           % The initial goal causes a rule to
   2: legal_furniture(LF)]           % to fire with introductory information.
  ==>                                % It will set a new goal.
  [retract(1),
   nl,
   write('Enter a single item of furniture at each prompt. '),nl,
   write('Include the width (in feet) of each item. '),nl,
   write('The format is Item:Length. '),nl,nl,
   write('The legal values are: '),nl,
   write(LF),nl,nl,
```

```

write('When there is no more furniture, enter "end:end."',nl,
assert(goal(read_furniture))].

rule 2:
[1: furniture(end,end),           % When the furniture is read
 2: goal(read_furniture)]         % set the new goal of reading
==>                               % reading wall sizes
[retract(all),
 assert(goal(read_walls))].

rule 3:
[1: goal(read_furniture),         % Loop to read furniture.
 2: legal_furniture(LF)]
==>
[prompt('furniture> ', F:L),
 member(F,LF),
 assert(furniture(F,L))].

rule 4:
[1: goal(read_furniture),         % If rule 3 matched and failed
 2: legal_furniture(LF)]         % the action, then member must
==>                               % have failed.
[write('Unknown piece of furniture, must be one of:'),nl,
 write(LF),nl].

rule 5:
[1: goal(read_walls)]
==>
[retract(1),
 prompt('What is the length of the north and south sides? ', LengthNS),
 prompt('What is the length of the east and west sides? ', LengthEW),
 assert(wall(north,LengthNS)),
 assert(wall(south,LengthNS)),
 assert(wall(east,LengthEW)),
 assert(wall(west,LengthEW)),
 assert(goal(find_door))].

rule 6:
[1: goal(find_door)]
==>
[retract(1),
 prompt('Which wall has the door? ', DoorWall),
 prompt('What is the width of the door? ', DoorWidth),
 retract(wall(DoorWall,X)),
 NewWidth = X - DoorWidth,
 assert(wall(DoorWall, NewWidth)),
 assert(position(door,DoorWall)),
 assert(goal(find_plugs)),
 write('Which walls have plugs? "end." when no more plugs:'),nl].

rule 7:
[1: goal(find_plugs),
 2: position(plug,end)]
==>
[retract(all)].

rule 8:
[1: goal(find_plugs)]
==>
[prompt('Side: ', Wall),
 assert(position(plug,Wall))].

```

```

% Rules f1-f13 illustrate the strength of rule based programming.
% Each rule captures a rule of thumb used in configuring furniture
% in a living room. The rules are all independent, transparent,
% and can be easily maintained. Complexity can be added without
% concern for the flow of control.

% f1, f2 - place the couch first, it should be either opposite the
% door, or to its right, depending on which wall is longer.

rule f1:
  [1: furniture(couch,LenC),           % an unplaced couch
   position(door, DoorWall),          % find the wall with the door
   opposite(DoorWall, OW),            % the wall opposite the door
   right(DoorWall, RW),               % the wall to the right of the door
  2: wall(OW, LenOW),                 % available space opposite
   wall(RW, LenRW),                  % available space to the right
   LenOW >= LenRW,                   % if opposite wall bigger than right
   LenC =< LenOW]                     % length of couch less than wall space
==>
  [retract(1),                         % remove the furniture term
   assert(position(couch, OW)),        % assert the new position
   retract(2),                         % remove the old wall,length
   NewSpace = LenOW - LenC,            % calculate the space now available
   assert(wall(OW, NewSpace))].       % assert the wall with new space left

rule f2:
  [1: furniture(couch,LenC),
   2: position(door, DoorWall),
   3: opposite(DoorWall, OW),
   4: right(DoorWall, RW),
   5: wall(OW, LenOW),
   6: wall(RW, LenRW),
   LenOW =< LenRW,
   LenC =< LenRW]
==>
  [retract(1),
   assert(position(couch, RW)),
   retract(6),
   NewSpace = LenRW - LenC,
   assert(wall(RW, NewSpace))].

% f3 - the tv should be opposite the couch

rule f3:
  [1: furniture(tv,LenTV),
   2: position(couch, CW),
   3: opposite(CW, W),
   4: wall(W, LenW),
   LenW >= LenTV]
==>
  [retract(1),
   assert(position(tv, W)),
   retract(4),
   NewSpace = LenW - LenTV,
   assert(wall(W, NewSpace))].

% f4, f5 - the coffee table should be in front of the couch or if there
% is no couch, in front of a chair.

rule f4:
  [1: furniture(coffee_table,_),
   2: position(couch, CW)]
==>

```

```

[retract(1),
 assert(position(coffee_table, front_of_couch:CW))].

rule f5:
[1: furniture(coffee_table,_),
 2: position(chair, CW)]
==>
[retract(1),
 assert(position(coffee_table, front_of_chair:CW))].

% f6, f7 - chairs should be on adjacent walls from the couch

rule f6:
[1: furniture(chair,LC),
 position(couch, CW),
 right(CW, ChWa),
 left(CW, ChWb),
 4: wall(ChWa, La),
 wall(ChWb, Lb),
 La >= Lb,
 La >= LC]
==>
[retract(1),
 assert(position(chair, ChWa)),
 NewSpace = La - LC,
 retract(4),
 assert(wall(ChWa, NewSpace))].

rule f7:
[1: furniture(chair,LC),
 position(couch, CW),
 right(CW, ChWa),
 left(CW, ChWb),
 wall(ChWa, La),
 4: wall(ChWb, Lb),
 La =< Lb,
 Lb >= LC]
==>
[retract(1),
 assert(position(chair, ChWb)),
 NewSpace = Lb - LC,
 retract(4),
 assert(wall(ChWb, NewSpace))].

rule f8:
[1: furniture(chair,LC),
 2: position(couch, CW),
 3: left(CW, ChW),
 4: wall(ChW, L),
 L >= LC]
==>
[retract(1),
 assert(position(chair, ChW)),
 NewSpace = L - LC,
 retract(4),
 assert(wall(ChW, NewSpace))].

% put end_tables next to the couch first, then on the walls with
% the chairs

rule f9:
[1: furniture(end_table,TL),
 2: position(couch, W),

```

```

3: not(position(end_table, W)),
4: wall(W, L),
   L >= TL]
==>
[retract(1),
 assert(position(end_table, W, nolamp)),
 NewSpace = L - TL,
 retract(4),
 assert(wall(W, NewSpace))].

rule f10:
[1: furniture(end_table, TL),
 2: position(chair, W),
 3: not(position(end_table, W)),
 4: wall(W, L),
   L >= TL]
==>
[retract(1),
 assert(position(end_table, W, nolamp)),
 NewSpace = L - TL,
 retract(4),
 assert(wall(W, NewSpace))].

% put the table lamps on the end tables

rule f11:
[1: furniture(table_lamp, _),
 2: position(end_table, W, nolamp)]
==>
[retract(all),
 assert(position(table_lamp, W)),
 assert(position(end_table, W, lamp))].

% get extension cords if needed

rule f12:
[1: position(tv, W),
 2: not(position(plug, W))]
==>
[assert(buy(extension_cord, W)),
 assert(position(plug, W))].

rule f13:
[1: position(table_lamp, W),
 2: not(position(plug, W))]
==>
[assert(buy(extension_cord, W)),
 assert(position(plug, W))].

% When no other rules fire, here is the summary

rule f14:
[1: not_end_yet]
==>
[retract(1),
 write('Recommendations:'), nl, nl,
 write('furniture positions:'), nl, nl,
 list(position(_, _)),
 list(position(_, _, _)), nl,
 write('purchase recommendations:'), nl, nl,
 list(buy(_, _)), nl,
 write('furniture which wouldn't fit:'), nl, nl,
 list(furniture(_, _)), nl, nl].

```

Animal Knowledgebase (animal.okb)

```
% from Winston & Horn's LISP

% Rules for animal identification. The first three rules are an
% input loop. Enter attributes that match the patterns in the rules.
% For example: has(robie,hair), or lays_eggs(suzie). These facts will
% help identify robie and suzie. Enter "end" to end the input loop.
%
% The attributes can also be put in the list of initial_data.
% Example:
%
%   initial_data([has(dennis,hair),
%                 has(dennis,hoofs),
%                 has(dennis,black_stripes),
%                 parent(dennis,diana)
%                 ]
%               ).
%
% This should lead to the identification of dennis and diana as zebras.

initial_data([goal(animal_id)]).

rule 1:
[1: goal(animal_id)]
==>
[assert(read_facts),
 retract(1)].

rule 2:
[1: end,
 2: read_facts]
==>
[retract(all)].

rule 3:
[1: read_facts]
==>
[prompt('Attribute ? ',X),
 assert(X)].

rule id1:
[1: has(X,hair)]
==>
[assert(isa(X,mammal)),
 retract(all)].

rule id2:
[1: gives(X,milk)]
==>
[assert(isa(X,mammal)),
 retract(all)].

rule id3:
[1: has(X,feathers)]
==>
[assert(isa(X,bird)),
 retract(all)].
```

```

rule id4:
  [1: flies(X),
   2: lays_eggs(X)]
  ==>
  [assert(isa(X,bird)),
   retract(all)].

rule id5:
  [1: eats_meat(X)]
  ==>
  [assert(isa(X,carnivore)),
   retract(all)].

rule id6:
  [1: has(X,pointed_teeth),
   2: has(X,claws),
   3: has(X,forward_eyes)]
  ==>
  [assert(isa(X,carnivore)),
   retract(all)].

rule id7:
  [1: isa(X,mammal),
   2: has(X,hoofs)]
  ==>
  [assert(isa(X,ungulate)),
   retract(all)].

rule id8:
  [1: isa(X,mammal),
   2: chews_cud(X)]
  ==>
  [assert(isa(X,ungulate)),
   assert(even_toed(X)),
   retract(all)].

rule id9:
  [1: isa(X,mammal),
   2: isa(X,carnivore),
   3: has(X,tawny_color),
   4: has(X,dark_spots)]
  ==>
  [assert(isa(X,cheetah)),
   retract(all)].

rule id10:
  [1: isa(X,mammal),
   2: isa(X,carnivore),
   3: has(X,tawny_color),
   4: has(X,black_stripes)]
  ==>
  [assert(isa(X,tiger)),
   retract(all)].

rule id11:
  [1: isa(X,ungulate),
   2: has(X,long_neck),
   3: has(X,long_legs),
   4: has(X,dark_spots)]

```

```
==>
[assert(isa(X,giraffe)),
 retract(all)].

rule id12:
[1: isa(X,ungulate),
 2: has(X,black_stripes)]
==>
[assert(isa(X,zebra)),
 retract(all)].

rule id13:
[1: isa(X,bird),
 2: does_not_fly(X),
 3: has(X,long_neck),
 4: has(X,long_legs),
 5: has_attr(X,black_and_white)]
==>
[assert(isa(X,ostrich)),
 retract(all)].

rule id14:
[1: isa(X,bird),
 2: does_not_fly(X),
 3: swims(X),
 4: has_attr(X,black_and_white)]
==>
[assert(isa(X,penguin)),
 retract(all)].

rule id15:
[1: isa(X,bird),
 2: flies_well(X)]
==>
[assert(isa(X,albatross)),
 retract(all)].

rule id16:
[1: isa(Animal,Type),
 2: parent(Animal,Child)]
==>
[assert(isa(Child,Type)),
 retract(all)].

rule id17:
[1: even_toed(X),
 2: has_attr(X,slow),
 3: isa(X,ungulate)]
==>
[assert(isa(X,sloth)),
 retract(all)].
```

Oops Interpreter (oops.pro)

```
% OOPS2 - A toy production system interpreter. It uses a forward chaining,
%       data driven, rule based approach for expert system development.
%
% Version 2, the simplest version without LEX, MEA, or conflict sets
%
% author Dennis Merritt
% Copyright (c) Dennis Merritt, 1986

% operator definitions

:-op(800,xfx,==>).      % used to separate LHS and RHS of rule
:-op(500,xfy,:).       % used to separate attributes and values
:-op(810,fx,rule).     % used to define rule
:-op(700,xfy,#).      % used for unification instead of =

main :-
    welcome,
    supervisor.

welcome :-
    nl, nl,
    write($      OOPS - A Toy Production System$), nl, nl,
    write($This is an interpreter for files containing rules coded in the$), nl,
    write($OOPS format.$), nl, nl,
    write($The => prompt accepts three commands:$), nl, nl,
    write($  load. -  prompts for name of rules file$), nl,
    write($      enclose in single quotes$), nl,
    write($  list. -  lists working memory$), nl,
    write($  go.  -  starts the inference$), nl,
    write($  exit. -  does what you'd expect$), nl, nl.

% the supervisor, uses a repeat fail loop to read and process commands
% from the user

supervisor :-
    repeat,
        write('=>'),
        read(X),
%     write(echo1-X),
        doit(X),
%     write(echo2-X),
    X = exit.

doit(X) :-
    do(X).

% actions to take based on commands

do(exit) :-
    !.
do(go) :-
    initialize,
    go,
    !.
do(load) :-
    load,
    !.
do(list) :-
    lst,      % lists all of working storage
```

```

!.
do(list(X)) :-
  lst(X),      % lists all which match the pattern
  !.
do(_) :-
  write('invalid command').

% loads the rules (Prolog terms) into the Prolog database

load :-
  write('Enter file name in single quotes (ex. 'room.okb'.): '),
  read(F),
  reconsult(F).      % loads a rule file into interpreter work space

% assert each of the initial conditions into working storage

initialize :-
  initial_data(X),
  assert_list(X).

% working storage is represented by database terms stored
% under the key "fact"

assert_list([]) :-
  !.
assert_list([H|T]) :-
  assertz(fact(H)),
  !,
  assert_list(T).

% the main inference loop, find a rule and try it.  if it fired, say so
% and repeat the process.  if not go back and try the next rule.  when
% no rules succeed, stop the inference

go :-
  call(rule ID: LHS ==> RHS),
  try(LHS,RHS),
  write('Rule fired '), write(ID), nl,
  !,
  go.
go.

% find the current conflict set.

%conflict_set(CS) :-
%  bagof(rule ID: LHS ==> RHS,
%        [rule ID: LHS ==> RHS, match(LHS)],CS).

% match the LHS against working storage, if it succeeds, process the
% actions from the RHS

try(LHS,RHS) :-
  match(LHS),
  process(RHS,LHS),
  !.

% recursively go through the LHS list, matching conditions against
% working storage

match([]) :-
  !.

```

```

match([N:Prem|Rest]) :-
    !,
    (fact(Prem)
     ;
     test(Prem)           % a comparison test rather than a fact
    ),
    match(Rest).
match([Prem|Rest]) :-
    (fact(Prem)           % condition number not specified
     ;
     test(Prem)
    ),
    match(Rest).

% various tests allowed on the LHS

test(not(X)) :-
    fact(X),
    !,
    fail.
test(not(X)) :-
    !.
test(X # Y) :-
    X = Y,
    !.
test(X > Y) :-
    X > Y,
    !.
test(X >= Y) :-
    X >= Y,
    !.
test(X < Y) :-
    X < Y,
    !.
test(X =< Y) :-
    X =< Y,
    !.
test(X = Y) :-
    X is Y,
    !.
test(member(X,Y)) :-
    member(X,Y),
    !.

% recursively execute each of the actions in the RHS list

process([],_) :-
    !.
process([Action|Rest],LHS) :-
    take(Action,LHS),
    !,
    process(Rest,LHS).

% if its retract, use the reference numbers stored in the Lrefs list,
% otherwise just take the action

take(retract(N),LHS) :-
    (N == all
     ;
     integer(N)
    ),
    retr(N,LHS),!.
take(A,_) :-
    take(A),!.

```

```

take(retract(X)) :-
    retract(fact(X)),
    !.
take(assert(X)) :-
    asserta(fact(X)),
    write(adding-X), nl,
    !.
take(X # Y) :-
    X = Y,
    !.
take(X = Y) :-
    X is Y,
    !.
take(write(X)) :-
    write(X),
    !.
take(nl) :-
    nl,
    !.
take(read(X)) :-
    read(X),
    !.
take(prompt(X,Y)) :-
    nl, write(X), read(Y),
    !.
take(member(X,Y)) :-
    member(X,Y),
    !.
take(list(X)) :-
    lst(X),
    !.

% logic for retraction

retr(all,LHS) :-
    retrall(LHS),
    !.
retr(N,[]) :-
    write('retract error, no '-N), nl,
    !.
retr(N,[N:Prem|_]) :-
    retract(fact(Prem)),
    !.
retr(N,[_|Rest]) :-
    !,
    retr(N,Rest).

retrall([]).
retrall([N:Prem|Rest]) :-
    retract(fact(Prem)),
    !, retrall(Rest).
retrall([Prem|Rest]) :-
    retract(fact(Prem)),
    !, retrall(Rest).
retrall(_|Rest) :-      % must have been a test
    retrall(Rest).

% list all of the terms in working storage

lst :-
    fact(X),
    write(X), nl,
    fail.
lst :-
    !.

```

```
% lists all of the terms which match the pattern

lst(X) :-
    fact(X),
    write(X), nl,
    fail.
lst(_) :-
    !.

% utilities

member(X, [X|_]).
member(X, [_|Z]) :-
    member(X, Z).
```


D Foops

Room Knowledgebase (room.fkb)

```
% ROOM.FKB - a version of the room knowledge base for FOOPS. Much of the
% knowledge about furniture is stored in frames, thus simplifying
% the rule portion of the knowledge base.
```

```
frame(furniture, [
    legal_types - [val [couch,chair,coffee_table,end_table,standing_lamp,
        table_lamp,tv,knickknack]],
    position - [def none, add pos_add],
    length - [def 3],
    place_on - [def floor],
    can_hold - [def 0]]).
```

```
frame(couch, [
    ako - [val furniture],
    length - [def 6]]).
```

```
frame(chair, [
    ako - [val furniture],
    length - [def 3]]).
```

```
% A table is different from most furniture in that it can hold things
% on it.
```

```
frame(table, [
    ako - [val furniture],
    space - [def 4],
    length - [def 4],
    can_support - [def yes],
    holding - [def []]]).
```

```
frame(end_table, [
    ako - [val table],
    length - [def 2]]).
```

```
frame(coffee_table, [
    ako - [val table],
    length - [def 4]]).
```

```
% electric is used as a super class for anything electrical. It contains
% the defaults for those attributes unique to electrical things.
```

```
frame(electric, [
    needs_outlet - [def yes]]).
```

```
frame(lamp, [
    ako - [val [furniture, electric]]]).
```

```
frame(standing_lamp, [
    ako - [val lamp]]).
```

```
frame(table_lamp, [
    ako - [val lamp],
    place_on - [def table]]).
```

```
frame(tv, [
    ako - [val [furniture, electric]],
    place_on - [calc tv_support]]).
```

```

frame(knickknack, [
    ako - [val furniture],
    length - [def 1],
    place_on - [def table]]).

frame(wall, [
    length - [def 10],
    outlets - [def 0],
    space - [calc space_calc],
    holding - [def []]]).

frame(door, [
    ako - [val furniture],
    length - [def 4]]).

frame(goal, []).

frame(recommend, []).

% calculate the available space if needed. The available space is
% computed from the length of the item minus the sum of the lengths of
% the items it is holding. The held items are in the holding list.
% The items in the list are identified only by their unique names.
% This is used by walls and tables.

space_calc(C,N,space-S) :-
    getf(C,N,[length-L,holding-HList]),
    sum_lengths(HList,0,HLen),
    S is L - HLen.

sum_lengths([],L,L).

sum_lengths([C/N|T],X,L) :-
    getf(C,N,[length-HL]),
    XX is X + HL,
    sum_lengths(T,XX,L).

% When placing the tv, check with the user to see if it goes on the
% floor or a table.

tv_support(tv,N,place_on-table) :-
    nl,
    write('Should the TV go on a table? '),
    read(yes),
    uptf(tv,N,[place_on-table]).

tv_support(tv,N,place_on-floor) :-
    uptf(tv,N,[place_on-floor]).

% Whenever a piece is placed in position, update the holding list of the
% item which holds it (table or wall) and the available space. If something
% is placed in front of something else, then do nothing.

pos_add(_,_,position-frontof(X)) :-
    uptf(C,N,[holding-[X]]).

pos_add(C,N,position-CP/P) :-
    getf(CP,P,[space-OldS]),
    getf(C,N,[length-L]),
    NewS is OldS - L,
    NewS >= 0,
    uptf(CP,P,[holding-[C/N],space-NewS]).

```

```

pos_add(C,N,position-CP/P) :-
    nl, write_line(['Not enough room on',CP,P,for,C,N]),
    !, fail.

% The forward chaining rules of the system. They make use of call
% to activate some pure Prolog predicates at the end of the knowledge
% base. In particular, data gathering, and wall space calculations
% are done in Prolog.

% These are the terms which are initially stored in working storage.
% They set a goal used to force firing of certain preliminary rules,
% and various facts about the problem domain used by the actual
% configuration rules.

initial_data([goal - gather_data,
    wall - north with [opposite-south,right-west,left-east],
    wall - south with [opposite-north,right-east,left-west],
    wall - east with [opposite-west,right-north,left-south],
    wall - west with [opposite-east,right-south,left-north] ]).

% first gather data, then try the couch first.

rule 1:
    [goal - gather_data]
    ==>
    [call(gather_data),
    assert( goal - couch_first )].

% Rules f1-f13 illustrate the strength of rule based programming.
% Each rule captures a rule of thumb used in configuring furniture
% in a living room. The rules are all independent, transparent,
% and can be easily maintained. Complexity can be added without
% concern for the flow of control.

% f1, f2 - place the couch first, it should be either opposite the
% door, or to its right, depending on which wall has more space.

rule f1:
    [goal - couch_first,
    couch - C with [position-none,length-LenC],
    door - D with [position-wall/W],
    wall - W with [opposite-OW,right-RW],
    wall - OW with [space-SpOW],
    wall - RW with [space-SpRW],
    SpOW >= SpRW,
    LenC =< SpOW]
    ==>
    [update(couch - C with [position-wall/OW])].

rule f2:
    [goal - couch_first,
    couch - C with [position-none,length-LenC],
    door - D with [position-wall/W],
    wall - W with [opposite-OW,right-RW],
    wall - OW with [space-SpOW],
    wall - RW with [space-SpRW],
    SpRW >= SpOW,
    LenC =< SpRW]
    ==>
    [update(couch - C with [position-wall/RW])].

% f3 - f3a the tv should be opposite the couch. if it needs a table, an
% end table should be placed under it, if no table is available put

```

```
% it on the floor anyway and recommend the purchase of a table. The rules
% first check to see if the couch has been placed.
```

```
rule f3:
  [couch - C with [position-wall/W],
   wall - W with [opposite-OW],
   tv - TV with [position-none,place_on-floor]]
  ==>
  [update(tv - TV with [position-wall/OW])].
```

```
rule f4:
  [couch - C with [position-wall/W],
   wall - W with [opposite-OW],
   tv - TV with [position-none,place_on-table],
   end_table - T with [position-none]]
  ==>
  [update(end_table - T with [position-wall/OW]),
   update(tv - TV with [position-end_table/T])].
```

```
rule f4a:
  [tv - TV with [position-none,place_on-table]]
  ==>
  [assert(recommend - R with [buy-['table for tv']])].
```

```
% f5 - the coffee table should be in front of the couch.
```

```
rule f5:
  [coffee_table - CT with [position-none],
   couch - C]
  ==>
  [update(coffee_table - CT with [position-frontof(couch/C)])].
```

```
% f6, f7 - chairs should be on adjacent walls from the couch, which ever
% has the most space
```

```
rule f6:
  [chair - Ch with [position-none],
   couch - C with [position-wall/W],
   wall - W with [right-RW,left-LW],
   wall - RW with [space-SpR],
   wall - LW with [space-SpL],
   SpR >= SpL]
  ==>
  [update(chair - Ch with [position-wall/RW])].
```

```
rule f7:
  [chair - Ch with [position-none],
   couch - C with [position-wall/W],
   wall - W with [right-RW,left-LW],
   wall - RW with [space-SpR],
   wall - LW with [space-SpL],
   SpL > SpR]
  ==>
  [update(chair - Ch with [position-wall/LW])].
```

```
% put end_tables next to the couch first, then on the walls with
% the chairs
```

```
rule f9:
  [end_table - ET with [position-none],
   not tv - TV with [position-none,place_on-table],
   couch - C with [position-wall/W],
```

```

    not end_table - ET2 with [position-wall/W]
    ==>
    [update(end_table - ET with [position-wall/W])].

rule f10:

    [end_table - ET with [position-none],
    not tv - TV with [position-none,place_on-table],
    chair - C with [position-wall/W],
    not end_table - ET2 with [position-wall/W]]
    ==>
    [update(end_table - ET with [position-wall/W])].

% put the table lamps on the end tables

rule f11:
    [table_lamp - TL with [position-none],
    end_table - ET with [position-wall/W]]
    ==>
    [update( table_lamp - TL with [position-end_table/ET] )].

% put the knickknacks on anything which will hold them.

rule f11a:
    [knickknack - KK with [position-none],
    Table - T with [can_support=yes, position-wall/W]]
    ==>
    [update( knickknack - KK with [position-Table/T] )].

% get extension cords if needed

rule f12:
    [Thing - X with [needs_outlet=yes, position-wall/W],
    wall - W with [outlets=0]]
    ==>
    [assert(recommend - R with [buy-['extension cord'-W]])].

rule f13:
    [Thing - X with [needs_outlet=yes, position-C/N],
    C - N with [position-wall/W],
    wall - W with [outlets=0]]
    ==>
    [assert(recommend - R with [buy-['extension cord'-Thing/W]])].

% When no other rules fire, here is the summary

rule f14:
    []
    ==>
    [call(output_data)].

% Prolog predicates called by various rules to perform functions better
% handled by Prolog.

% Gather the input data from the user.

gather_data :-
    read_furniture,
    read_walls.

```

```

read_furniture :-
    get_frame(furniture,[legal_types-LT]),
    write('Enter name of furniture at the prompt. It must be one of:'), nl,
    write(LT), nl,
    write('Enter 'end.' to stop input. '), nl,
    write('At the length prompt enter 'y.' or a new number. '), nl,
    repeat,
    write('>'), read(X),
    process_furn(X),
    !.

process_furn(end).
process_furn(X) :-
    get_frame(X,[length-DL]),
    write(length-DL),write('>'),
    read(NL),
    get_length(NL,DL,L),
    addf(X,_,[length-L]),
    fail.

get_length(y,L,L) :- !.
get_length(L,_,L).

read_walls :-
    nl, write('Enter data for the walls. '), nl,
    write('What is the length of the north & south walls? '),
    read(NSL),
    uptf(wall,north,[length-NSL]),
    uptf(wall,south,[length-NSL]),
    write('What is the length of the east & west walls? '),
    read(EWL),
    uptf(wall,east,[length-EWL]),
    uptf(wall,west,[length-EWL]),
    write('Which wall has the door? '),
    read(DoorWall),
    write('What is its length? '),
    read(DoorLength),
    addf(door,D,[length-DoorLength]),
    uptf(door,D,[position-wall/DoorWall]),
    write('Which walls have outlets? (a list)'),
    read(PlugWalls),
    process_plugs(PlugWalls).

process_plugs([]) :- !.
process_plugs([H|T]) :-
    uptf(wall,H,[outlets-1]),
    !,
    process_plugs(T).
process_plugs(X) :-
    uptf(wall,X,[outlets-1]).

output_data :-
    write('The final results are:'),nl,
%    print_frames,
    output_walls,
    output_tables,
    output_recommends,
    output_unplaced.

output_walls :-
    getf(wall,W,[holding-HL]),
    write_line([W,wall,holding|HL]),

```

```
    fail.
output_walls.

output_tables :-
    getf(C,N,[holding-HL]),
    not C = wall,
    write_line([C,N,holding|HL]),
    fail.
output_tables.

output_recommends :-
    getf(recommend,_,[buy-BL]),
    write_line([purchase|BL]),
    fail.
output_recommends.

output_unplaced :-
    write('Unplaced furniture:'), nl,
    getf(T,N,[position-none]),
    write(T-N), nl,
    fail.
output_unplaced.
```

Foops (foops.pro)

```
% FOOPS.PRO - an integration of frames, forward chaining with LEX and MEA,
% and Prolog.
% Copyright (c) Dennis Merritt, 1986 - Permission granted for
% non-commercial use

% The first section of the code contains the basic OOPS code, the
% second section contains the FRAMES code.

% OPERATOR DEFINITIONS

:-op(800,xfx,==>).           % used to separate LHS and RHS of rule
:-op(500,xfy,:).             % used to separate attributes and values
:-op(810,fx,rule).           % used to define rule
:-op(700,xfy,#).             % used for unification instead of =
:-op(700,xfy,\=).            % not equal
:-op(600,xfy,with).          % used for frame instances in rules

main :-
    welcome,
    supervisor.

welcome :-
    write($FOOPS - A Toy Production System$),nl,nl,
    write($This is an interpreter for files containing rules coded in the$),nl,
    write($FOOPS format.$), nl, nl,
    write($The => prompt accepts four commands:$), nl, nl,
    write($  load. - prompts for name of rules file$), nl,
    write($      enclose in single quotes$), nl,
    write($  go.  - starts the inference$), nl,
    write($  list. - list working memory$), nl,
    write($  exit. - does what you'd expect$), nl, nl.

% THE SUPERVISOR, USES A REPEAT FAIL LOOP TO READ AND PROCESS COMMANDS
% FROM THE USER

supervisor :-
    repeat,
    write('=>'),
    read(X),
    doit(X),
    X = exit.

doit(X) :- do(X).

% ACTIONS TO TAKE BASED ON COMMANDS

do(exit) :- !.
do(go) :-
    initialize,
    timer(T1),
    go,
    timer(T2),
    T is 10 * (T2 - T1),
    write(time-T), nl, !.
do(load) :- load, !.
do(list) :- lst, !.           % lists all of working storage
do(list(X)) :- lst(X), !.    % lists all which match the pattern
do(_) :- write('invalid command'), nl.
```

```

% LOADS THE RULES (PROLOG TERMS) INTO THE PROLOG DATABASE

load :-
    write('Enter the file name in single quotes (ex. 'room.fkb'.): '),
    read(F),
    reconsult(F).          % loads a rule file into interpreter work space

% ASSERT EACH OF THE INITIAL CONDITIONS INTO WORKING STORAGE

initialize :-
    setchron(1),
    abolish(instantiation,1),
    delf(all),
    assert(mea(no)),
    assert(gid(100)),
    initial_data(X),
    assert_list(X), !.
initialize :-
    error(301,[initialization,error]).

% WORKING STORAGE IS REPRESENTED BY DATABASE TERMS STORED
% UNDER THE KEY "fact"

assert_list([]) :- !.
assert_list([H|T]) :-
    getchron(Time),
    assert_ws( fact(H,Time) ),
    !, assert_list(T).

% THE MAIN INFERENCE LOOP, FIND A RULE AND TRY IT.  IF IT FIRED, SAY SO
% AND REPEAT THE PROCESS.  IF NOT GO BACK AND TRY THE NEXT RULE.  WHEN
% NO RULES SUCCEED, STOP THE INFERENCE

go :-
    conflict_set(CS),
    write_cs(CS),
    select_rule(CS,r(Inst,ID,LHS,RHS)),
    write($Rule Selected $), write(ID), nl,
    (process(RHS,LHS); true),
    asserta( instantiation(Inst) ),
    write($Rule fired $), write(ID), nl,
    !, go.
go.

write_cs([]).
write_cs([r(I,ID,L,R)|X]) :-
    write(ID), nl,
    writeinst(I),
    write_cs(X).

writeinst([]).
writeinst([H|T]) :-
    tab(5),
    write(H), nl,
    writeinst(T).

conflict_set(CS) :-
    bagof(r(Inst,ID,LHS,RHS),
        (rule ID: LHS ==> RHS, match(LHS,Inst)), CS).

```

```

select_rule(CS,R) :-
    refract(CS,CS1),
    mea_filter(0,CS1,[],CSR),
    lex_sort(CSR,R).

list_cs([]).
list_cs([K-r(_ ,ID,_,_)|T]) :-
    write(ID-K), nl,
    list_cs(T).

% ELIMINATE those rules which have already been tried

refract([],[]).
refract([r(Inst,_,_,_)|T],TR) :-
    instantiation(Inst),
    !, refract(T,TR).
refract([H|T],[H|TR]) :-
    refract(T,TR).

% SORT THE REST OF THE CONFLICT SET ACCORDING TO THE LEX STRATEGY

lex_sort(L,R) :-
    build_keys(L,LK),
% keysort(LK,X),
    sort(LK,X),
    reverse(X,[K-R|_]).

% BUILD LISTS OF TIME STAMPS FOR LEX SORT KEYS

build_keys([],[]).
build_keys([r(Inst,A,B,C)|T],[Key-r(Inst,A,B,C)|TR]) :-
    build_chlist(Inst,ChL),
    sort(ChL,X),
    reverse(X,Key),
    build_keys(T,TR).

% BUILD A LIST OF JUST THE TIMES OF THE VARIOUS MATCHED ATTRIBUTES
% FOR USE IN RULE SELECTION

build_chlist([],[]).
build_chlist([_ /Chron|T],[Chron|TC]) :-
    build_chlist(T,TC).

% ADD THE TEST FOR MEA IF APPROPRIATE THAT EMPHASIZES THE FIRST ATTRIBUTE
% SELECTED.

mea_filter(_ ,X,_,X) :- not mea(yes), !.
mea_filter(_ ,[],X,X).
mea_filter(Max,[r([A/T|Z],B,C,D)|X],Temp,ML) :-
    T < Max,
    !, mea_filter(Max,X,Temp,ML).
mea_filter(Max,[r([A/T|Z],B,C,D)|X],Temp,ML) :-
    T = Max,
    !, mea_filter(Max,X,[r([A/T|Z],B,C,D)|Temp],ML).
mea_filter(Max,[r([A/T|Z],B,C,D)|X],Temp,ML) :-
    T > Max,
    !, mea_filter(T,X,[r([A/T|Z],B,C,D)],ML).

% RECURSIVELY GO THROUGH THE LHS LIST, MATCHING CONDITIONS AGAINST
% WORKING STORAGE

```

```

match([], []).
match([Prem|Rest],[Prem/Time|InstRest]) :-
    mat(Prem,Time),
    match(Rest,InstRest).

mat(N:Prem,Time) :-
    !, fact(Prem,Time).
mat(Prem,Time) :-
    fact(Prem,Time).
mat(Test,0) :-
    test(Test).

fact(Prem,Time) :-
    conv(Prem,Class,Name,ReqList),
    getf(Class,Name,ReqList,Time).

assert_ws( fact(Prem,Time) ) :-
    conv(Prem,Class,Name,UList),
    addf(Class,Name,UList).

update_ws( fact(Prem,Time) ) :-
    conv(Prem,Class,Name,UList),
    uptf(Class,Name,UList).

retract_ws( fact(Prem,Time) ) :-
    conv(Prem,Class,Name,UList),
    delf(Class,Name,UList).

conv(Class-Name with List, Class, Name, List).
conv(Class-Name, Class, Name, []).

% VARIOUS TESTS ALLOWED ON THE LHS

test(not(X)) :-
    fact(X,_),
    !, fail.
test(not(X)) :- !.
test(X#Y) :- X=Y, !.
test(X>Y) :- X>Y, !.
test(X>=Y) :- X>=Y, !.
test(X<Y) :- X<Y, !.
test(X<=Y) :- X<=Y, !.
test(X \= Y) :- not X=Y, !.
%test(X = Y) :- X=Y, !.
test(X = Y) :- X is Y, !.
test(is_on(X,Y)) :- is_on(X,Y), !.
test(call(X)) :- call(X).

% RECURSIVELY EXECUTE EACH OF THE ACTIONS IN THE RHS LIST

process([],_) :- !.
process([Action|Rest],LHS) :-
    take(Action,LHS),
    !, process(Rest,LHS).
process([Action|Rest],LHS) :-
    error(201,[Action,fails]).

% IF ITS RETRACT, USE THE REFERENCE NUMBERS STORED IN THE Lrefs LIST,
% OTHERWISE JUST TAKE THE ACTION

```

```

take(retract(N),LHS) :-
    (N == all; integer(N)),
    retr(N,LHS), !.
take(A,_ ) :- take(A), !.

take(retract(X)) :- retract_ws(fact(X,_)), !.
take(assert(X)) :-
    getchron(T),
    assert_ws(fact(X,T)),
    write(adding-X), nl,
    !.
take(update(X)) :-
    getchron(T),
    update_ws(fact(X,T)),
    write(updating-X), nl,
    !.
take(X # Y) :- X=Y, !.
take(X = Y) :- X is Y, !.
take(write(X)) :- write(X), !.
take(write_line(X)) :- write_line(X), !.
take(nl) :- nl, !.
take(read(X)) :- read(X), !.
take(prompt(X,Y)) :- nl, write(X), read(Y), !.
take(cls) :- cls, !.
take(is_on(X,Y)) :- is_on(X,Y), !.
take(list(X)) :- lst(X), !.
take(call(X)) :- call(X).

% LOGIC FOR RETRACTION

retr(all,LHS) :-retrall(LHS), !.
retr(N,[]) :- error(202,['retract error, no ',N]), !.
retr(N,[N:Prem|_]) :- retract_ws(fact(Prem,_)), !.
retr(N,[_|Rest]) :- !, retr(N,Rest).

retrall([]).
retrall([N:Prem|Rest]) :-
    retract_ws(fact(Prem,_)),
    !, retrall(Rest).
retrall([Prem|Rest]) :-
    retract_ws(fact(Prem,_)),
    !, retrall(Rest).
retrall([_|Rest]) :-      % must have been a test
    retrall(Rest).

% LIST ALL OF THE TERMS IN WORKING STORAGE

lst :-
    fact(X,_),
    write(X), nl,
    fail.
lst.

% LISTS ALL OF THE TERMS WHICH MATCH THE PATTERN

lst(X) :-
    fact(X,_),
    write(X), nl,
    fail.
lst(_).

```

```

% UTILITIES

member(X, [X|Y]).
member(X, [Y|Z]) :-
    member(X, Z).

reverse(F,R) :-
    rever(F, [], R).

rever([], R, R).
rever([X|Y], T, R) :-
    rever(Y, [X|T], R).

% MAINTAIN A TIME COUNTER

setchron(N) :-
    retract( chron(_ ) ),
    asserta( chron(N) ), !.
setchron(N) :-
    asserta( chron(N) ).

getchron(N) :-
    retract( chron(N) ),
    NN is N + 1,
    asserta( chron(NN) ), !.

%
% THIS SECTION IMPLEMENTS A FRAME BASED SCHEME FOR KNOWLEDGE REPRESENTATION
%

:- op(600, fy, val).
:- op(600, fy, calc).
:- op(600, fy, def).
:- op(600, fy, add).
:- op(600, fy, del).

% prep_req takes a request of the form Slot-Val, and forms it into the
% more accurate req(Class, Slot, Facet, Value). If no facet was mentioned
% in the original request, then the facet of "any" is used to indicate
% the system should use everything possible to find a value.

prep_req(Slot-X, req(C, N, Slot, val, X)) :- var(X), !.
prep_req(Slot-X, req(C, N, Slot, Facet, Val)) :-
    nonvar(X),
    X =.. [Facet, Val],
    facet_list(FL),
    is_on(Facet, FL), !.
prep_req(Slot-X, req(C, N, Slot, val, X)).

facet_list([val, def, calc, add, del, edit]).

% RETRIEVE A LIST OF SLOT VALUES

get_frame(Class, ReqList) :-
    frame(Class, SlotList),
    slot_vals(Class, _, ReqList, SlotList).

getf(Class, Name, ReqList) :-
    getf(Class, Name, ReqList, _).

```

```

getf(Class,Name,ReqList,TimeStamp) :-
    frinst(Class, Name, SlotList, TimeStamp),
    slot_vals(Class, Name, ReqList, SlotList).

slot_vals(_,_ ,ReqL,SlotL) :-
    var(ReqL),
    !,
    ReqL = SlotL.
slot_vals(_,_ ,[],_).
slot_vals(C,N,[Req|Rest],SlotList) :-
    prep_req(Req,req(C,N,S,F,V)),
    find_slot(req(C,N,S,F,V),SlotList),
    !, slot_vals(C,N,Rest,SlotList).
slot_vals(C,N, Req, SlotList) :-
    not(list(Req)),
    prep_req(Req,req(C,N,S,F,V)),
    find_slot(req(C,N,S,F,V), SlotList).

find_slot(req(C,N,S,F,V), SlotList) :-
    nonvar(V), !,
    find_slot(req(C,N,S,F,Val), SlotList), !,
    (Val = V; list(Val),is_on(V,Val)).
find_slot(req(C,N,S,F,V), SlotList) :-
    is_on(S-FacetList, SlotList), !,
    facet_val(req(C,N,S,F,V),FacetList).
find_slot(req(C,N,S,F,V), SlotList) :-
    is_on(ako-FacetList, SlotList),
    facet_val(req(C,N,ako,val,Ako),FacetList),
    (is_on(X,Ako); X = Ako),
    frame(X, HigherSlots),
    find_slot(req(C,N,S,F,V), HigherSlots), !.
find_slot(Req,_ ) :-
    error(99,['frame error looking for:',Req]).

facet_val(req(C,N,S,F,V),FacetList) :-
    FV =.. [F,V],
    is_on(FV,FacetList), !.
facet_val(req(C,N,S,val,V),FacetList) :-
    is_on(val ValList,FacetList),
    is_on(V,ValList), !.
facet_val(req(C,N,S,S,val,V),FacetList) :-
    is_on(calc Pred,FacetList),
    CalcPred =.. [Pred,C,N,S-V],
    call(CalcPred), !.
facet_val(req(C,N,S,S,val,V),FacetList) :-
    is_on(def V,FacetList), !.

% ADD A LIST OF SLOT VALUES

add_frame(Class, UList) :-
    old_slots(Class,SlotList),
    add_slots(Class,_ ,UList,SlotList,NewList),
    retract(frame(Class,_)),
    asserta(frame(Class,NewList)), !.

addf(Class,Nm,UList) :-
    (var(Nm),genid(Name);Name=Nm),
    add_slots(Class,Name,[ako-Class|UList],SlotList,NewList),
    getchron(TimeStamp),
    asserta( frinst(Class,Name,NewList,TimeStamp) ),
    !.

```

```

uptf(Class,Name,UList) :-
    frinst(Class,Name,SlotList,_),
    add_slots(Class,Name,UList,SlotList,NewList),
    retract( frinst(Class,Name,_,_) ),
    getchron(TimeStamp),
    asserta( frinst(Class,Name,NewList,TimeStamp) ),
    !.
uptf(Class,Name,UList) :-
    error(105,[update,failed,Class,Name,UList]).

genid(G) :-
    retract(gid(N)),
    G is N + 1,
    asserta(gid(G)).

old_slots(Class,SlotList) :-
    frame(Class,SlotList), !.
old_slots(Class,[]) :-
    asserta(frame(Class,[])).

add_slots(_,_,[],X,X).
add_slots(C,N,[U|Rest],SlotList,NewList) :-
    prep_req(U,req(C,N,S,F,V)),
    add_slot(req(C,N,S,F,V),SlotList,Z),
    !, add_slots(C,N,Rest,Z,NewList).
add_slots(C,N,X,SlotList,NewList) :-
    prep_req(X,req(C,N,S,F,V)),
    add_slot(req(C,N,S,F,V),SlotList,NewList).

add_slot(req(C,N,S,F,V),SlotList,[S-FL2|SL2]) :-
    delete(S-FacetList,SlotList,SL2),
    add_facet(req(C,N,S,F,V),FacetList,FL2).

add_facet(req(C,N,S,F,V),FacetList,[FNew|FL2]) :-
    FX =.. [F,OldVal],
    delete(FX,FacetList,FL2),
    add_newval(OldVal,V,NewVal),
    !, check_add_demons(req(C,N,S,F,V),FacetList),
    FNew =.. [F,NewVal].

add_newval(X,Val,Val) :- var(X), !.
add_newval(OldList,ValList,NewList) :-
    list(OldList),
    list(ValList),
    append(ValList,OldList,NewList), !.
add_newval([H|T],Val,[Val,H|T]).
add_newval(_,Val,Val).

check_add_demons(req(C,N,S,F,V),FacetList) :-
    get_frame(C,S-add(Add)), !,
    AddFunc =.. [Add,C,N,S-V],
    call(AddFunc).
check_add_demons(_,_).

% DELETE A LIST OF SLOT VALUES

del_frame(Class) :-
    retract(frame(Class,_)).
del_frame(Class) :-
    error(203,['No frame',Class,'to delete']).

```

```

del_frame(Class, UList) :-
    old_slots(Class, SlotList),
    del_slots(Class, _, UList, SlotList, NewList),
    retract(frame(Class, _)),
    asserta(frame(Class, NewList)).

delf(all) :-
    retract( frinst( _, _, _, _ ) ),
    fail.
delf(all).

delf(Class, Name) :-
    retract( frinst(Class, Name, _, _ ) ),
    !.
delf(Class, Name) :-
    error(103, ['No instance of ', Class, ' for ', Name]).

delf(Class, Name, UList) :-
    old_flots(Class, Name, SlotList),
    del_slots(Class, Name, UList, SlotList, NewList),
    retract( frinst(Class, Name, _, _ ) ),
    getchron(TimeStamp),
    asserta( frinst(Class, Name, NewList, TimeStamp) ).

del_slots( _, _, [], X, X ).
del_slots(C, N, [U|Rest], SlotList, NewList) :-
    prep_req(U, req(C, N, S, F, V)),
    del_slot(req(C, N, S, F, V), SlotList, Z),
    del_slots(C, N, Rest, Z, NewList).
del_slots(C, N, X, SlotList, NewList) :-
    prep_req(X, req(C, N, S, F, V)),
    del_slot(req(C, N, S, F, V), SlotList, NewList).

del_slot(req(C, N, S, F, V), SlotList, [S-FL2|SL2]) :-
    remove(S-FacetList, SlotList, SL2),
    del_facet(req(C, N, S, F, V), FacetList, FL2).
del_slot(Req, _, _) :-
    error(104, ['del_slot - unable to remove', Req]).

del_facet(req(C, N, S, F, V), FacetList, FL) :-
    FV =.. [F, V],
    remove(FV, FacetList, FL),
    !, check_del_demons(req(C, N, S, F, V), FacetList).
del_facet(req(C, N, S, F, V), FacetList, [FNew|FL]) :-
    FX =.. [F, OldVal],
    remove(FX, FacetList, FL),
    remove(V, OldVal, NewValList),
    FNew =.. [F, NewValList],
    !, check_del_demons(req(C, N, S, F, V), FacetList).
del_facet(Req, _, _) :-
    error(105, ['del_facet - unable to remove', Req]).

check_del_demons(req(C, N, S, F, V), FacetList) :-
    get_frame(C, S-del(Del)), !,
    DelFunc =.. [Del, C, N, S-V],
    call(DelFunc).
check_del_demons( _, _ ).

```

```

% PRINT A FRAME

print_frames :-
    frame(Class, SlotList),
    print_frame(Class),
    fail.
print_frames.

print_frame(Class) :-
    frame(Class, SlotList),
    write_line(['Frame:', Class]),
    print_slots(SlotList), nl.

printfs :-
    frame(Class, _),
    printf(Class, _),
    fail.
printfs.

printf(Class, Name) :-
    frinst(Class, Name, SlotList, Time),
    write_line(['Frame:', Class, Name, Time]),
    print_slots(SlotList), nl.

printf(Class) :-
    frinst(Class, Name, SlotList, Time),
    write_line(['Frame:', Class, Name, Time]),
    print_slots(SlotList), nl, fail.
printf(_).

print_slots([]).
print_slots([Slot|Rest]) :-
    write_line([' Slot:', Slot]),
    print_slots(Rest).

% UTILITIES

delete(X, [], []).
delete(X, [X|Y], Y) :- !.
delete(X, [Y|Z], [Y|W]) :- delete(X, Z, W).

remove(X, [X|Y], Y) :- !.
remove(X, [Y|Z], [Y|W]) :- remove(X, Z, W).

is_on(X, [X|Y]).
is_on(X, [Y|Z]) :- is_on(X, Z).

error_threshold(100).

error(NE, _) :- error_threshold(N), N > NE, !, fail.
error(NE, E) :-
    nl, write('*** '), write(error-NE), tab(1),
    write_line(E),
    !, fail.

write_line([]) :- nl.

```

```
write_line([H|T]) :-  
    write(H), tab(1),  
    write_line(T).
```

```
time_test :-  
    write('TT> '),  
    read(X),  
    timer(T1),  
    X,  
    timer(T2),  
    nl, nl,  
    T is T2 - T1,  
    write(time-T).
```

E Rete-Foops

Room Knowledgebase (room.rkb)

```
% ROOM.RKB - a version of ROOM for use with RETE-FOOPS.

frame(furniture, [legal_types - [val [couch,chair,coffee_table,end_table,
                                   standing_lamp,table_lamp,tv,knickknack]
                               ],
               position - [def none, add pos_add],
               length - [def 3],
               place_on - [def floor],
               can_hold - [def 0]
               ]
).

frame(couch, [ako - [val furniture],
             length - [def 6]
             ]
).

frame(chair, [ako - [val furniture],
             length - [def 3]
             ]
).

% A table is different from most furniture in that it can hold things
% on it.

frame(table, [ako - [val furniture],
             space - [def 4],
             length - [def 4],
             can_support - [def yes],
             holding - [def []]
             ]
).

frame(end_table, [ako - [val table],
                 length - [def 2]
                 ]
).

frame(coffee_table, [ako - [val table],
                    length - [def 4]
                    ]
).

% electric is used as a super class for anything electrical. It contains
% the defaults for those attributes unique to electrical things.

frame(electric, [needs_outlet - [def yes]]).

frame(lamp, [ako - [val [furniture, electric]]]).

frame(standing_lamp, [ako - [val lamp]]).
```

```

frame(table_lamp, [ako - [val lamp],
                  place_on - [def table]
                  ]
      ).

frame(tv, [ako - [val [furniture, electric]],
          place_on - [calc tv_support]]
      ).

frame(knickknack, [ako - [val furniture],
                  length - [def 1],
                  place_on - [def table]
                  ]
      ).

frame(wall, [length - [def 10],
            outlets - [def 0],
            space - [calc space_calc],
            holding - [def []]
            ]
      ).

frame(door, [ako - [val furniture],
            length - [def 4]
            ]
      ).

frame(goal, []).

frame(recommend, []).

% Calculate the available space if needed. The available space is
% computed from the length of the item minus the sum of the lengths of
% the items it is holding. The held items are in the holding list.
% The items in the list are identified only by their unique names.
% This is used by walls and tables.

space_calc(C,N,space-S) :-
    getf(C,N,[length-L,holding-HList]),
    sum_lengths(HList,0,HLen),
    S is L - HLen.

sum_lengths([],L,L).
sum_lengths([C/N|T],X,L) :-
    getf(C,N,[length-HL]),
    XX is X + HL,
    sum_lengths(T,XX,L).

% When placing the tv, check with the user to see if it goes on the
% floor or a table.

tv_support(tv,N,place_on-table) :-
    nl,
    write('Should the TV go on a table? '),
    read(yes),
    uptf(tv,N,[place_on-table]).
tv_support(tv,N,place_on-floor) :-
    uptf(tv,N,[place_on-floor]).

```

```
% Whenever a piece is placed in position, update the holding list of the
% item which holds it (table or wall) and the available space. If something
% is placed in front of something else, then do nothing.
```

```
pos_add(_,_,position-frontof(X)) :-
    uptf(C,N,[holding-[X]]).
pos_add(C,N,position-CP/P) :-
    getf(CP,P,[space-OldS]),
    getf(C,N,[length-L]),
    NewS is OldS - L,
    NewS >= 0,
    uptf(CP,P,[holding-[C/N],space-NewS]).
pos_add(C,N,position-CP/P) :-
    nl,write_line(['Not enough room on',CP,P,for,C,N]),
    !,fail.
```

```
% The forward chaining rules of the system. They make use of call
% to activate some pure Prolog predicates at the end of the knowledge
% base. In particular, data gathering, and wall space calculations
% are done in Prolog.
```

```
% These are the terms that are initially stored in working storage.
% They set a goal used to force firing of certain preliminary rules,
% and various facts about the problem domain used by the actual
% configuration rules.
```

```
initial_data([
    wall - north with [opposite-south,right-west,left-east],
    wall - south with [opposite-north,right-east,left-west],
    wall - east with [opposite-west,right-north,left-south],
    wall - west with [opposite-east,right-south,left-north],
    goal - door_first,
    door - d1 with [length - 3],
    couch - c1 with [length - 6],
    chair - ch1 with [length - 3],
    chair - ch2 with [length - 3],
    chair - ch3 with [length - 3],
    chair - ch4 with [length - 3],
    chair - ch5 with [length - 3],
    chair - ch6 with [length - 3],
    chair - ch7 with [length - 3],
    tv - tv1 with [length - 2, place_on - floor] ]).
```

```
% first gather data, then try the couch first.
```

```
rule 1#
    [goal - gather_data]
    ==>
    [call(gather_data),
     assert( goal - couch_first )].
```

```
rule a1#
    [goal - door_first]
    ==>
    [update( door - d1 with [position - wall/east]),
     assert( goal - couch_first )].
```

```
% Rules f1-f13 illustrate the strength of rule based programming.
% Each rule captures a rule of thumb used in configuring furniture
% in a living room. The rules are all independent, transparent,
% and can be easily maintained. Complexity can be added without
% concern for the flow of control.
```

```

% f1, f2 - place the couch first, it should be either opposite the
% door, or to its right, depending on which wall has more space.

rule f1#
[goal - couch_first,
 couch - C with [position-none,length-LenC],
 door - D with [position-wall/W],
 wall - W with [right-RW]]
==>
[update(couch - C with [position-wall/RW])].

rule f2#
[goal - couch_first,
 couch - C with [position-none,length-LenC],
 door - D with [position-wall/W],
 wall - W with [opposite-OW]]
==>
[update(couch - C with [position-wall/OW])].

% f3 - f3a the tv should be opposite the couch. If it needs a table, an
% end table should be placed under it, if no table is available put
% it on the floor anyway and recommend the purchase of a table. The rules
% first check to see if the couch has been placed.

rule f3#
[couch - C with [position-wall/W],
 wall - W with [opposite-OW],
 tv - TV with [position-none,place_on-floor]]
==>
[update(tv - TV with [position-wall/OW])].

rule f4#
[couch - C with [position-wall/W],
 wall - W with [opposite-OW],
 tv - TV with [position-none,place_on-table],
 end_table - T with [position-none]]
==>
[update(end_table - T with [position-wall/OW]),
 update(tv - TV with [position-end_table/T])].

rule f4a#
[tv - TV with [position-none,place_on-table]]
==>
[assert(recommend - R with [buy-['table for tv']])].

% f5 - the coffee table should be in front of the couch.

rule f5#
[coffee_table - CT with [position-none],
 couch - C]
==>
[update(coffee_table - CT with [position-frontof(couch/C)])].

% f6, f7 - chairs should be on adjacent walls from the couch, which ever
% has the most space

rule f6#
[chair - Ch with [position-none],
 couch - C with [position-wall/W],
 wall - W with [right-RW]]
==>

```

```

[update(chair - Ch with [position-wall/RW])].

rule f7#
  [chair - Ch with [position-none],
   couch - C with [position-wall/W],
   wall - W with [left-LW]]
  ==>
  [update(chair - Ch with [position-wall/LW])].

% put end_tables next to the couch first, then on the walls with
% the chairs

%rule f9#
% [end_table - ET with [position-none],
% not tv - TV with [position-none,place_on-table],
% couch - C with [position-wall/W],
% not end_table - ET2 with [position-wall/W]]
% ==>
% [update(end_table - ET with [position-wall/W])].

%rule f10#
% [end_table - ET with [position-none],
% not tv - TV with [position-none,place_on-table],
% chair - C with [position-wall/W],
% not end_table - ET2 with [position-wall/W]]
% ==>
% [update(end_table - ET with [position-wall/W])].

% put the table lamps on the end tables

rule f11#
  [table_lamp - TL with [position-none],
   end_table - ET with [position-wall/W]]
  ==>
  [update( table_lamp - TL with [position-end_table/ET] )].

% put the knickknacks on anything which will hold them.

%rule f11a#
% [knickknack - KK with [position-none],
% Table - T with [can_support-yes, position-wall/W]]
% ==>
% [update( knickknack - KK with [position-Table/T] )].

% get extension cords if needed

%rule f12#
% [Thing - X with [needs_outlet-yes, position-wall/W],
% wall - W with [outlets-0]]
% ==>
% [assert(recommend - R with [buy-['extension cord'-W]])].

%rule f13#
% [Thing - X with [needs_outlet-yes, position-C/N],
% C - N with [position-wall/W],
% wall - W with [outlets-0]]
% ==>
% [assert(recommend - R with [buy-['extension cord'-Thing/W]])].

% When no other rules fire, here is the summary

```

```

finished :-
    output_data.

% Prolog predicates called by various rules to perform functions better
% handled by Prolog.

% Gather the input data from the user.

gather_data :-
    read_furniture,
    read_walls.

read_furniture :-
    get_frame(furniture,[legal_types-LT]),
    write('Enter name of furniture at the prompt. It must be one of:'), nl,
    write(LT), nl,
    write('Enter end to stop input. '), nl,
    write('At the length prompt enter y or a new number. '), nl,
    repeat,
        write('>'), read(X),
        process_furn(X),
    !. % end was input

process_furn(end).
process_furn(X) :-
    get_frame(X,[length-DL]),
    write(length-DL), write('>'),
    read(NL),
    get_length(NL,DL,L),
    assert_ws(X - _ with [length-L]),
    fail.

get_length(y,L,L) :- !.
get_length(L,_ ,L).

read_walls :-
    nl, write('Enter data for the walls. '), nl,
    write('What is the length of the north & south walls? '),
    read(NSL),
    update_ws(wall-north with [length-NSL]),
    update_ws(wall-south with [length-NSL]),
    write('What is the length of the east & west walls? '),
    read(EWL),
    update_ws(wall-east with [length-EWL]),
    update_ws(wall-west with [length-EWL]),
    write('Which wall has the door? '),
    read(DoorWall),
    write('What is its length? '),
    read(DoorLength),
    assert_ws(door-D with [length-DoorLength]),
    update_ws(door-D with [position-wall/DoorWall]),
    write('Which walls have outlets? (a list)'),
    read(PlugWalls),
    process_plugs(PlugWalls).

process_plugs([]) :- !.
process_plugs([H|T]) :-
    update_ws(wall-H with [outlets-1]),
    !,
    process_plugs(T).
process_plugs(X) :-

```

```

update_ws(wall-X with [outlets-1]).

output_data :-
    write('The final results are:'), nl,
    output_walls,
    output_tables,
    output_recommends,
    output_unplaced.

output_walls :-
    getf(wall,W,[holding-HL]),
    write_line([W,wall,holding|HL]),
    fail.
output_walls.

output_tables :-
    getf(C,N,[holding-HL]),
    not C = wall,
    write_line([C,N,holding|HL]),
    fail.
output_tables.

output_recommends :-
    getf(recommend,_,[buy-BL]),
    write_line([purchase|BL]),
    fail.
output_recommends.

output_unplaced :-
    write('Unplaced furniture:'), nl,
    getf(T,N,[position-none]),
    write(T-N), nl,
    fail.
output_unplaced.

```

Rete Compiler (retepred.pro)

```
% RETEPRED.PRO - the predicates that implement the Rete pattern
%                  matching algorithm.

% It should be modified some day to use pointers to working memory in
% the memory predicates rather than the full tokens - this would save
% a lot of space.

% retecomp - compile rules into a rete network

:-op(800,xfx,==>).      % used to separate LHS and RHS of rule
:-op(500,xfy,#).       % used to separate attributes and values
:-op(810,fx,rule).     % used to define rule
:-op(700,xfy,#).       % used for unification instead of =
:-op(700,xfy,\=).     % not equal
:-op(600,xfy,with).   % used for frame instances in rules

rete_compile :-
    abolish(root,3),
    abolish(bi,4),
    abolish(tes,4),
    abolish(rul,3),
    abolish(varg,1),
    abolish(nid,1),
    asserta(nid(0)),
    rete_compil.
% display_net.

display_net :-
    display_roots,nl,
    display_bis,nl,
    display_teses,nl,
    display_ruls.

display_roots :-
    root(N,A,B),
    write( root(N,A,B) ), nl,
    fail.
display_roots.

display_bis :-
    bi(A,B,C,D),
    write(bi(A)), nl,
    write_list([left|B]),
    write_list([right|C]),
    write(D), nl, nl,
    fail.
display_bis.

display_teses :-
    tes(A,B,C,D),
    write(tes(A)), nl,
    write_list([left|B]),
    write_list([right|C]), nl,
    write(D), nl, nl,
    fail.
display_teses.

display_ruls :-
    rul(A,B,C),
```

```

    write(rul(A)), nl,
    write_list([left|B]),
    write_list([right|C]), nl,
    fail.
display_ruls.

write_list([]).
write_list([H|T]) :-
    write(H), nl,
    wr_lis(T).

wr_lis([]).
wr_lis([H|T]) :-
    tab(5),write(H), nl,
    wr_lis(T).

% compile each rule into the rete net

rete_compil :-
    rule N# LHS ==> RHS,
    rete_comp(N,LHS,RHS),
    fail.
rete_compil :-
    message(201).

% compile an individual rule into the net

rete_comp(N, [H|T],RHS) :-
    term(H,Hw),
    check_root(RN,Hw,HList),
    retcom(root(RN), [Hw/_],HList,T,N,RHS),
    message(202,N),
    !.
rete_comp(N,_,_) :-
    message(203,N).

% the main compile loop
% PNID - the id of the previous node
% OutTok - list of tokens from previous node
% PrevList - transfer list from previous node
% [H|T] - list of remaining clauses in rule
% N - The rule ID, for building the rule at the end
% RHS - the rhs of the rule for building the rule at the end

retcom(PNID,OutTok,PrevList,[],N,RHS) :-
    build_rule(OutTok,PrevList,N,RHS),
    update_node(PNID,PrevList,rule-N),
    !.
retcom(PNID,PrevNode,PrevList,[H|T],N,RHS) :-
    term(H,Hw),
    check_root(RN,Hw,HList),
    check_node(PrevNode,PrevList,[Hw/_],HList,NID,OutTok,NList),
    update_node(PNID,PrevList,NID-1),
    update_root(RN,HList,NID-r),
    !,
    retcom(NID,OutTok,NList,T,N,RHS).
retcom(PNID,PrevNode,PrevList,[H|T],N,RHS) :- % some kind of tester call
    check_tnode(PrevNode,PrevList,[H/0],HList,NID,OutTok,NList),
    update_node(PNID,PrevList,test-NID),
    !,
    retcom(test-NID,OutTok,NList,T,N,RHS).

term(Class-Name with List,Class-Name with List).

```

```

term(Class-Name, Class-Name with []).

check_root(NID,Term,[]) :-
    not(root(_,Term,_)),
    gen_nid(NID),
    assertz( root(NID,Term,[]) ),
    !.
check_root(N,Term,List) :-
    asserta(temp(Term)),
    retract(temp(T1)),
    root(N,Term,List),
    root(N,T2,_),
    comp_devar(T1,T2),
    !.
check_root(NID,Term,[]) :-
    gen_nid(NID),
    assertz( root(NID,Term,[]) ).

% if this node was already on the list do nothing, otherwise add it
% to the list

update_root(RN,HList,NID) :-
    member(NID,HList),
    !.
update_root(RN,HList,NID) :-
    retract(root(RN,H,HList)),
    asserta(root(RN,H,[NID|HList])).

update_node(root(RN),HList,NID) :-
    update_root(RN,HList,NID),
    !.
update_node(X,PrevList,NID) :-
    member(NID,PrevList),
    !.
update_node(test-N,PrevList,NID) :-
    retract(tes(N,L,T,_)),
    asserta(tes(N,L,T,[NID|PrevList])),
    !.
update_node(PNID,PrevList,NID) :-
    retract(bi(PNID,L,R,_)),
    asserta(bi(PNID,L,R,[NID|PrevList])).

% check to see if there is a node which already fits, otherwise
% create a new one
% PNode - token list from previous node
% PList - list of successor nodes from previous node
% H - new token being added
% HList - successor nodes from root for token H
% NID - returned ID of the node
% OutTok - returned tokenlist from the node
% NList - returned list of successor nodes from the node

% first case - there isn't a matching rule using Prolog's match, so
% build a new one

check_node(PNode,PList,H,HList,NID,OutTok,[]) :-
    not(bi(_,PNode,H,_)),
    append(PNode,H,OutTok),
    gen_nid(NID),
    assertz(bi(NID,PNode,H,[])),
    !.

% second case - there was a matching rule using Prolog's match, so

```

```

% match again using generated constants instead of variables. If
% this matches then we have a match, otherwise we had a match
% where variables don't line up and its no good. (asserts and
% retracts allow different variables to have same information and
% prevent binding of variables in one from affecting the other)

check_node(PNode,PList,H,HList,NID,OutTok,NList) :-
    append(PNode,H,OutTok),
    asserta(temp(OutTok)),
    retract(temp(Tot1)),
    bi(NID,PNode,H,NList),
    bi(NID,T2,T3,_),
    append(T2,T3,Tot2),
    comp_devar(Tot1,Tot2),
    !.

% third case - the variables didn't line up from the second rule, so
% make a new node.

check_node(PNode,PList,H,HList,NID,OutTok,[]) :-
    append(PNode,H,OutTok),
    gen_nid(NID),
    assertz(bi(NID,PNode,H,[])).

% check for test node - similar to check for regular node

check_tnode(PNode,PList,H,HList,NID,OutTok,[]) :-
    not(tes(_,PNode,H,_)),
    append(PNode,H,OutTok),
    gen_nid(NID),
    assertz(tes(NID,PNode,H,[])),
    !.

% second case - there was a matching rule using Prolog's match, so
% match again using generated constants instead of variables. If
% this matches then we have a match, otherwise we had a match
% where variables don't line up and its no good. (asserts and
% retracts allow different variables to have same information and
% prevent binding of variables in one from affecting the other)

check_tnode(PNode,PList,H,HList,NID,OutTok,NList) :-
    append(PNode,H,OutTok),
    asserta(temp(OutTok)),
    retract(temp(Tot1)),
    tes(NID,PNode,H,NList),
    tes(NID,T2,T3,_),
    append(T2,T3,Tot2),
    comp_devar(Tot1,Tot2),
    !.

% third case - the variables didn't line up from the second rule, so
% make a new node.

check_tnode(PNode,PList,H,HList,NID,OutTok,[]) :-
    append(PNode,H,OutTok),
    gen_nid(NID),
    assertz(tes(NID,PNode,H,[])).

build_rule(OutTok,PrevList,N,RHS) :-
    assertz(rul(N,OutTok,RHS)).

gen_nid(NID) :-

```

```

    retract(nid(N)),
    NID is N + 1,
    asserta(nid(NID)).

% the hard part, undo Prolog's pattern matching so variables match just
% variables and not constants. de-var replaces all the variables with
% generated constants - this ensures only variables will match variables.

comp_devar(T1,T2) :-
    de_vari(T1),
    de_vari(T2),
    T1 = T2.

de_vari([]).
de_vari([H|T]) :-
    de_var(H),
    de_vari(T).
de_vari(X) :-
    de_var(X).

de_var(X/_ ) :-
    de_var(X).
de_var(X-Y with List) :-
    init_vargen,
    de_v(X-Y),
    de_vl(List),
    !.
de_var(X-Y) :-
    init_vargen,
    de_v(X-Y),
    !.

de_vl([]).
de_vl([H|T]) :-
    de_v(H),
    de_vl(T).

de_v(X-Y) :-
    d_v(X),
    d_v(Y).

d_v(V) :-
    var(V),
    var_gen(V),
    !.
d_v(_).

init_vargen :-
    abolish(varg,1),
    asserta(varg(1)).

var_gen(V) :-
    retract(varg(N)),
    NN is N + 1,
    asserta(varg(NN)),
    string_integer(NS,N),
    string_list(NS,NL),
    append("#VAR_",NL,X),
    name(V,X).

```

```

% predicates to update the rete network

% add a token to the rete net.  a token is of the form C-N with [S-V,...]
% ReqList gets bound with the values from the term added to the database.

addrete(Class,Name,TimeStamp) :-
    root(ID,Class-Name with ReqList, NextList),
    ffsend(Class,Name,ReqList,TimeStamp,NextList),
    fail.
addrete(_,_,_).

% fullfill the request list from the token, and send the instantiated
% token through the net.

ffsend(Class,Name,ReqList,TimeStamp,NextList) :-
    getf(Class,Name,ReqList),
    send(tok(add,[(Class-Name with ReqList)/TimeStamp]), NextList),
    !.

delrete(Class,Name,TimeStamp) :-
    root(ID,Class-Name with ReqList, NextList),
    delr(Class,Name,ReqList,TimeStamp),
    fail.
delrete(_,_,_).

delr(Class,Name,ReqList,TimeStamp) :-
    getf(Class,Name,ReqList),
    !,
    send(tok(del,[(Class-Name with ReqList)/TimeStamp]), NextList).
delr(Class,Name,ReqList,TimeStamp).

% send the new token to each of the sucesor nodes

send(_, []).
send(Tokens, [Node|Rest]) :-
    sen(Node, Tokens),
    send(Tokens, Rest).

% add or delete the new token from the appropriate memory, build new
% tokens from left or right and send them to successor nodes.

sen(rule-N, tok(AD,TokenList)) :-
    rul(N,TokenList,Actions),
    (AD = add, add_conflict_set(N,TokenList,Actions);
     AD = del, del_conflict_set(N,TokenList,Actions)
    ),
    !.
sen(Node-l, tok(AD,TokenList)) :-
    bi(Node,TokenList,Right,NextList),
    (AD = add, asserta(memory(Node-l,TokenList));
     AD = del, retract(memory(Node-l,TokenList))
    ),
    !,
    matchRight(Node,AD,TokenList,Right,NextList).
sen(Node-r, tok(AD,TokenList)) :-
    bi(Node,Left,TokenList,NextList),
    (AD = add, asserta(memory(Node-r,TokenList));
     AD = del, retract(memory(Node-r,TokenList))
    ),
    !,
    matchLeft(Node,AD,TokenList,Left,NextList).
sen(test-N, tok(AD,TokenList)) :-

```

```
tes(N,TokenList,[Test/0],NextList),
test(Test),
append(TokenList,[Test/0],NewToks),
!,
send(tok(AD,NewToks),NextList).

matchRight(Node,AD,TokenList,Right,NextList) :-
memory(Node-r,Right),
append(TokenList,Right,NewToks),
send(tok(AD,NewToks),NextList),
fail.
matchRight(_,_,_,_,_).

matchLeft(Node,AD,TokenList,Left,NextList) :-
memory(Node-l,Left),
append(Left,TokenList,NewToks),
send(tok(AD,NewToks),NextList),
fail.
matchLeft(_,_,_,_,_).
```

Rete Runtime (retefoop.pro)

```
% RETEFOOP.PRO - forward chaining, frames, and Rete algorithm, also using
%                 LEX and MEA to sort the conflict set.
%
% Copyright (c) Dennis Merritt, 1988

% operator definitions

:-op(800,xfx,==>).           % used to separate LHS and RHS of rule
:-op(500,xfy,#).            % used to separate attributes and values
:-op(810,fx,rule).          % used to define rule
:-op(700,xfy,#).            % used for unification instead of =
:-op(700,xfy,\=).           % not equal
:-op(600,xfy,with).         % used for frame instances in rules

main :- welcome, supervisor.

welcome :-
    write($      RETEFOOP - A Toy Production System$), nl, nl,
    write($This is an interpreter for files containing rules coded in the$), nl,
    write($FOOPS format.$), nl, nl,
    write($The => prompt accepts three commands:$), nl, nl,
    write($  load.      - prompts for name of rules file$), nl,
    write($              enclose in single quotes$), nl,
    write($  compile.   - compiles rules into a rete net$), nl,
    write($  displaynet. - displays the rete net$), nl,
    write($  list.      - lists stuff$), nl,
    write($  list(X).    - lists things which match X$), nl,
    write($  options.   - allows setting of message levels$), nl,
    write($  go.        - starts the inference$), nl,
    write($  exit.      - does what you'd expect$), nl, nl.

% the supervisor, uses a repeat fail loop to read and process commands
% from the user

supervisor :-
    repeat,
    write('=>'),
    read(X),
    doit(X),
    X = exit.

doit(X) :-
    timer(T1),
    do(X),
    timer(T2),
    T is (T2 - T1) / 600,
    message(101,T),
    !.

% actions to take based on commands

do(exit) :- !.
do(go) :-
    initialize,
    go,
    !.
do(load) :-
    load,
    !.
```

```

do(compile) :-
    compile,
    !.
do(displaynet) :-
    display_net,
    !.
do(list) :-
    lst, % lists all of working storage
    !.
do(list(X)) :-
    lst(X), % lists all that match the pattern
    !.
do(options) :-
    set_messtypes,
    !.
do(_) :-
    message(102).

% loads the rules (Prolog terms) into the Prolog database

load :-
    write('Enter the file name in single quotes (ex. ''room.rkb'') : '),
    read(F),
    reconsult(F), % loads a rule file into interpreter work space
    rete_compile. % ** rete change **

compile :-
    rete_compile.

% assert each of the initial conditions into working storage

initialize :-
    message(120),
    abolish(memory,2),
    abolish(inst,3),
    setchron(1),
    delf(all),
    abolish(conflict_set,1),
    assert(conflict_set([])),
    assert(mea(no)),
    initial_data(X),
    assert_list(X),
    message(121),
    !.
initialize :-
    message(103).

% working storage is represented frame instances - frinsts and also
% stored in a rete net

assert_list([]) :- !.
assert_list([H|T]) :-
    assert_ws(H),
    !,
    assert_list(T).

% the main inference loop, find a rule and try it. If it fired, say so
% and repeat the process. If not, go back and try the next rule. When
% no rules succeed, stop the inference.

go :-
    conflict_set(CS),
    select_rule(CS,inst(ID,LHS,RHS)),

```

```

    message(104, ID),
    (process(ID, RHS, LHS); true), % action side might fail
    del_conflict_set(ID, LHS, RHS),
    !,
    go.
go :-
    conflict_set([]),
    finished, % supplied in kb for what to do at end
    !.
go :-
    message(119).

del_conflict_set(N, TokenList, Action) :-
    conflict_set(CS),
    remove(inst(N, TokenList, Action), CS, CS2),
    message(105, N),
    retract(conflict_set(_)),
    asserta(conflict_set(CS2)).
del_conflict_set(N, TokenList, Action) :-
    message(106, N).

add_conflict_set(N, TokenList, Action) :-
    message(107, N),
    retract(conflict_set(CS)),
    asserta(conflict_set([inst(N, TokenList, Action) | CS])).

select_rule(CS, R) :-
    message(122, CS),
    mea_filter(0, CS, [], CSR),
    lex_sort(CSR, R).

% sort the rest of the conflict set according to the lex strategy

lex_sort(L, R) :-
    build_keys(L, LK),
    sort(LK, X),
    reverse(X, [K-R|_]).

% build lists of time stamps for lex sort keys

build_keys([], []).
build_keys([inst(N, TokenList, C) | T], [Key-inst(N, TokenList, C) | TR]) :-
    build_chlist(TokenList, ChL),
    sort(ChL, X),
    reverse(X, Key),
    build_keys(T, TR).

% build a list of just the times of the various matched attributes
% for use in rule selection

build_chlist([], []).
build_chlist([_ / Chron | T], [Chron | TC]) :-
    build_chlist(T, TC).

% add the test for mea if appropriate that emphasizes the first attribute
% selected.

mea_filter(_, X, _, X) :-
    not mea(yes),
    !.
mea_filter(_, [], X, X).
mea_filter(Max, [inst(N, [A/T|Z], C) | X], Temp, ML) :-

```

```

    T < Max,
    !,
    mea_filter(Max,X,Temp,ML) .
mea_filter(Max,[inst(N,[A/T|Z],C)|X],Temp,ML) :-
    T = Max,
    !,
    mea_filter(Max,X,[inst(N,[A/T|Z],C)|Temp],ML) .
mea_filter(Max,[inst(N,[A/T|Z],C)|X],Temp,ML) :-
    T > Max,
    !,
    mea_filter(T,X,[inst(N,[A/T|Z],C)],ML) .

get_ws(Prem,Time) :-
    conv(Prem,Class,Name,ReqList),
    getf(Class,Name,ReqList,Time) .

assert_ws(Prem) :-
    message(109,Prem),
    conv(Prem,Class,Name,AList),
    addf(Class,Name,AList,TimeStamp),
    addrete(Class,Name,TimeStamp) .

update_ws(Prem) :-
    conv(Prem,Class,Name,UList),
    frinst(Class,Name,_,TS),
    uptrf(Class,Name,UList,TimeStamp),    % note - does delrete in uptrf
    addrete(Class,Name,TimeStamp),
    !.
update_ws(Prem) :-
    message(108,Prem) .

retract_ws(Prem/T) :- retract_ws(Prem) .
retract_ws(Prem) :-
    conv(Prem,Class,Name,UList),
    delrete(Class,Name,TimeStamp),
    delf(Class,Name,UList) .

conv(Class-Name with List, Class, Name, List) .
conv(Class-Name, Class, Name, []).

% various tests allowed on the LHS

test(not(X)) :-
    get_ws(X,_),
    !,
    fail.
test(not(X)) :- !.
test(X#Y) :-
    X = Y,
    !.
test(X>Y) :-
    X > Y,
    !.
test(X>=Y) :-
    X >= Y,
    !.
test(X<Y) :-
    X < Y,
    !.
test(X<=Y) :-
    X <= Y,
    !.

```

```

test(X \= Y) :-
    not X = Y,
    !.
test(X = Y) :-
    X = Y,
    !.
test(X = Y) :-
    X is Y,
    !.
test(is_on(X,Y)) :-
    is_on(X,Y),
    !.
test(call(X)) :-
    call(X).

% recursively execute each of the actions in the RHS list

process(N, [], _) :-
    message(118,N),
    !.
process(N, [Action|Rest], LHS) :-
    take(Action, LHS),
    !,
    process(N, Rest, LHS).
process(N, [Action|Rest], LHS) :-
    message(110,N),
    !,
    fail.

% if its retract, use the reference numbers stored in the Lrefs list,
% otherwise just take the action

take(retract(N), LHS) :-
    (N == all; integer(N)),
    retr(N, LHS),
    !.
take(A, _) :-
    take(A),
    !.

take(retract(X)) :-
    retract_ws(X),
    !.
take(assert(X)) :-
    assert_ws(X),
    !.
take(update(X)) :-
    update_ws(X),
    !.
take(X # Y) :-
    X = Y,
    !.
take(X = Y) :-
    X is Y,
    !.
take(write(X)) :-
    write(X),
    !.
take(write_line(X)) :-
    write_line(X),
    !.
take(nl) :-
    nl,
    !.
take(read(X)) :-

```

```

    read(X),
    !.
take(prompt(X,Y)) :-
    nl,
    write(X),
    read(Y),
    !.
take(cls) :-
    cls,
    !.
take(is_on(X,Y)) :-
    is_on(X,Y),
    !.
take(list(X)) :-
    lst(X),
    !.
take(call(X)) :-
    call(X).

% logic for retraction

retr(all,LHS) :-
    retrall(LHS),
    !.
retr(N,[]) :-
    message(111,N),
    !.
retr(N,[N#Prem|_]) :-
    retract_ws(Prem),
    !.
retr(N,[_|Rest]) :-
    !,
    retr(N,Rest).

retrall([]).
retrall([N#Prem|Rest]) :-
    retract_ws(Prem),
    !,
    retrall(Rest).
retrall([Prem|Rest]) :-
    retract_ws(Prem),
    !,
    retrall(Rest).
retrall([_|Rest]) :- % must have been a test
    retrall(Rest).

% list all of the terms in working storage

lst :- printf$.

% lists all of the terms which match the pattern

lst(X) :-
    get_ws(X,_),
    write(X), nl,
    fail.
lst(_) :- !.

% maintain a time counter

setchron(N) :-
    retract(chron(_)),
    asserta(chron(N)),

```

```

!.
setchron(N) :-
    asserta(chron(N)).

getchron(N) :-
    retract(chron(N)),
    NN is N + 1,
    asserta(chron(NN)),
    !.

% this implements a frame based scheme for knowledge representation

:- op(600,fy,val).
:- op(600,fy,calc).
:- op(600,fy,def).
:- op(600,fy,add).
:- op(600,fy,del).

% prep_req takes a request of the form Slot-Val, and forms it into the
% more accurate req(Class,Slot,Facet,Value). If no facet was mentioned
% in the original request, then the facet of "any" is used to indicate
% the system should use everything possible to find a value.

prep_req(Slot-X, req(C,N,Slot,val,X)) :-
    var(X),
    !.
prep_req(Slot-X, req(C,N,Slot,Facet,Val)) :-
    nonvar(X),
    X =.. [Facet,Val],
    facet_list(FL),
    is_on(Facet,FL),
    !.
prep_req(Slot-X, req(C,N,Slot,val,X)).

facet_list([val,def,calc,add,del,edit]).

% retrieve a list of slot values

get_frame(Class, ReqList) :-
    frame(Class, SlotList),
    slot_vals(Class,_,ReqList,SlotList).

getf(Class,Name,ReqList) :-
    getf(Class,Name,ReqList,_).

getf(Class,Name,ReqList,TimeStamp) :-
    frinst(Class, Name, SlotList, TimeStamp),
    slot_vals(Class, Name, ReqList, SlotList).

slot_vals(_,_,[],_).
slot_vals(C,N,[Req|Rest],SlotList) :-
    prep_req(Req, req(C,N,S,F,V)),
    find_slot(req(C,N,S,F,V),SlotList),
    !,
    slot_vals(C,N,Rest,SlotList).
slot_vals(C,N,Req,SlotList) :-
    prep_req(Req, req(C,N,S,F,V)),
    find_slot(req(C,N,S,F,V),SlotList).

find_slot(req(C,N,S,F,V), SlotList) :-
    nonvar(V),

```

```

!,
find_slot(req(C,N,S,F,Val), SlotList),
!,
(Val = V; list(Val), is_on(V,Val)).
find_slot(req(C,N,S,F,V), SlotList) :-
is_on(S-FacetList, SlotList),
!,
facet_val(req(C,N,S,F,V),FacetList).
find_slot(req(C,N,S,F,V), SlotList) :-
is_on(ako-FacetList, SlotList),
facet_val(req(C,N,ako,val,Ako),FacetList),
(is_on(X,Ako); X = Ako),
frame(X, HigherSlots),
find_slot(req(C,N,S,F,V), HigherSlots),
!.
find_slot(Req,_) :-
message(112,Req),
fail.

facet_val(req(C,N,S,F,V),FacetList) :-
FV =.. [F,V],
is_on(FV,FacetList),
!.
facet_val(req(C,N,S,val,V),FacetList) :-
is_on(val ValList,FacetList),
is_on(V,ValList),
!.
facet_val(req(C,N,S,val,V),FacetList) :-
is_on(calc Pred,FacetList),
CalcPred =.. [Pred,C,N,S-V],
call(CalcPred),
!.
facet_val(req(C,N,S,val,V),FacetList) :-
is_on(def V,FacetList),
!.

% add a list of slot values

add_frame(Class, UList) :-
old_slots(Class,SlotList),
add_slots(Class,_,UList,SlotList,NewList),
retract(frame(Class,_)),
asserta(frame(Class,NewList)),
!.

addf(Class,Nm,UList) :-
addf(Class,Nm,UList,TimeStamp).

addf(Class,Nm,UList,TimeStamp) :-
(var(Nm), genid(Name); Name = Nm),
add_slots(Class,Name,[ako-Class|UList],SlotList,NewList),
getchron(TimeStamp),
asserta(frinst(Class,Name,NewList,TimeStamp)),
!.

uptf(Class,Name,UList) :-
uptf(Class,Name,UList,TS).

uptf(Class,Name,UList,TimeStamp) :-
frinst(Class,Name,SlotList,TS),
add_slots(Class,Name,UList,SlotList,NewList),
retract(frinst(Class,Name,_,_)),

```

```

    getchron(TimeStamp),
    asserta(frinst(Class,Name,NewList,TimeStamp)),
    !.
uptrf(Class,Name,UList,TimeStamp) :-
    message(113,[Class,Name,UList]).

uptrf(Class,Name,UList) :-
    uptrf(Class,Name,UList,TS).

uptrf(Class,Name,UList,TimeStamp) :-
    frinst(Class,Name,SlotList,TS),
    add_slots(Class,Name,UList,SlotList,NewList),
    delrete(Class,Name,TS),
    retract(frinst(Class,Name,_,_)),
    getchron(TimeStamp),
    asserta(frinst(Class,Name,NewList,TimeStamp)),
    !.
uptrf(Class,Name,UList,TimeStamp) :-
    message(113,[Class,Name,UList]).

genid(G) :-
    retract(gid(N)),
    G is N + 1,
    asserta(gid(G)).

gid(100).

old_slots(Class,SlotList) :-
    frame(Class,SlotList),
    !.
old_slots(Class,[]) :-
    asserta(frame(Class,[])).

old_flots(Class,Name,SlotList) :-
    frinst(Class,Name,SlotList,_).

add_slots(_,_,[],X,X).
add_slots(C,N,[U|Rest],SlotList,NewList) :-
    prep_req(U,req(C,N,S,F,V)),
    add_slot(req(C,N,S,F,V),SlotList,Z),
    !,
    add_slots(C,N,Rest,Z,NewList).
add_slots(C,N,X,SlotList,NewList) :-
    prep_req(X,req(C,N,S,F,V)),
    add_slot(req(C,N,S,F,V),SlotList,NewList).

add_slot(req(C,N,S,F,V),SlotList,[S-FL2|SL2]) :-
    delete(S-FacetList,SlotList,SL2),
    add_facet(req(C,N,S,F,V),FacetList,FL2).

add_facet(req(C,N,S,F,V),FacetList,[FNew|FL2]) :-
    FX =.. [F,OldVal],
    delete(FX,FacetList,FL2),
    add_newval(OldVal,V,NewVal),
    !,
    check_add_demons(req(C,N,S,F,V),FacetList),
    FNew =.. [F,NewVal].

```

```

add_newval(X,Val,Val) :-
    var(X),
    !.
add_newval(OldList,ValList,NewList) :-
    list(OldList),
    list(ValList),
    append(ValList,OldList,NewList),
    !.
add_newval([H|T],Val,[Val,H|T]).
add_newval(_,Val,Val).

check_add_demons(req(C,N,S,F,V),FacetList) :-
    get_frame(C,S-add(Add)),
    !,
    AddFunc =.. [Add,C,N,S-V],
    call(AddFunc).
check_add_demons(_,_).

% delete a list of slot values

del_frame(Class) :-
    retract(frame(Class,_)).
del_frame(Class) :-
    message(114,Class).

del_frame(Class, UList) :-
    old_slots(Class,SlotList),
    del_slots(Class,_,UList,SlotList,NewList),
    retract(frame(Class,_)),
    asserta(frame(Class,NewList)).

delf(all) :-
    retract(frinst(_,_,_,_)),
    fail.
delf(all).

delf(Class,Name) :-
    retract(frinst(Class,Name,_,_)),
    !.
delf(Class,Name) :-
    message(115,Class-Name).

delf(Class,Name,[]) :-
    !,
    delf(Class,Name).
delf(Class,Name,UList) :-
    old_flots(Class,Name,SlotList),
    del_slots(Class,Name,UList,SlotList,NewList),
    retract(frinst(Class,Name,_,_)),
    getchron(TimeStamp),
    asserta(frinst(Class,Name,NewList,TimeStamp)).

del_slots(_,_,[],X,X).
del_slots(C,N,[U|Rest],SlotList,NewList) :-
    prep_req(U,req(C,N,S,F,V)),
    del_slot(req(C,N,S,F,V),SlotList,Z),
    del_slots(C,N,Rest,Z,NewList).
del_slots(C,N,X,SlotList,NewList) :-
    prep_req(X,req(C,N,S,F,V)),
    del_slot(req(C,N,S,F,V),SlotList,NewList).

```

```

del_slot(req(C,N,S,F,V),SlotList,[S-FL2|SL2]) :-
    remove(S-FacetList,SlotList,SL2),
    del_facet(req(C,N,S,F,V),FacetList,FL2).
del_slot(Req,_,_) :-
    message(116,Req).

del_facet(req(C,N,S,F,V),FacetList,FL) :-
    FV =.. [F,V],
    remove(FV,FacetList,FL),
    !,
    check_del_demons(req(C,N,S,F,V),FacetList).
del_facet(req(C,N,S,F,V),FacetList,[FNew|FL]) :-
    FX =.. [F,OldVal],
    remove(FX,FacetList,FL),
    remove(V,OldVal,NewValList),
    FNew =.. [F,NewValList],
    !,
    check_del_demons(req(C,N,S,F,V),FacetList).
del_facet(Req,_,_) :-
    message(117,Req).

check_del_demons(req(C,N,S,F,V),FacetList) :-
    get_frame(C,S-del(Del)),
    !,
    DelFunc =.. [Del,C,N,S-V],
    call(DelFunc).
check_del_demons(_,_) .

% print a frame

print_frames :-
    frame(Class, SlotList),
    print_frame(Class),
    fail.
print_frames.

print_frame(Class) :-
    frame(Class,SlotList),
    write_line(['Frame:',Class]),
    print_slots(SlotList), nl.

printfs :-
    frame(Class,_),
    printf(Class,_),
    fail.
printfs.

printf(Class,Name) :-
    frinst(Class,Name,SlotList,Time),
    write_line(['Frame:',Class,Name,Time]),
    print_slots(SlotList), nl.

printf(Class) :-
    frinst(Class,Name,SlotList,Time),
    write_line(['Frame:',Class,Name,Time]),
    print_slots(SlotList), nl,
    fail.
printf(_).

```

```

print_slots([]).
print_slots([Slot|Rest]) :-
    write_line([' Slot:',Slot]),
    print_slots(Rest).

% utilities

delete(X, [], []).
delete(X, [X|Y], Y) :- !.
delete(X, [Y|Z], [Y|W]) :-
    delete(X, Z, W).

remove(X, [X|Y], Y) :- !.
remove(X, [Y|Z], [Y|W]) :-
    remove(X, Z, W).

is_on(X, [X|Y]).
is_on(X, [Y|Z]) :-
    is_on(X, Z).

write_line([]) :- nl.
write_line([H|T]) :-
    write(H), tab(1),
    write_line(T).

time_test :-
    write('TT> '),
    read(X),
    timer(T1),
    X,
    timer(T2),
    nl, nl,
    T is (T2 - T1) / 10,
    write(time-T).

append([H|T], W, [H|Z]) :-
    append(T, W, Z).
append([], W, W).

member(X, [X|_]).
member(X, [_|T]) :-
    member(X, T).

reverse(L1, L2) :-
    revzap(L1, [], L2).

revzap([X|L], L2, L3) :-
    revzap(L, [X|L2], L3).
revzap([], L, L).

% Message handling and messages

message(N) :-
    message(N, '').

message(N, Args) :-
    mess(N, break, Text),

```

```

    write(break), tab(1), write(N), write(': '), write(Text), write(Args), nl.
% break.
message(N,Args) :-
    mess(N,error,Text),
    write(error), tab(1), write(N), write(': '), write(Text), write(Args), nl,
    !,
    fail.
message(N,Args) :-
    mess(N,Type,Text),
    mess_types(TT),
    member(Type,TT),
    write(Type), tab(1), write(N), write(': '), write(Text), write(Args), nl,
    !.
message(_, _).

mess_types([info,trace,warning,debug]).

set_messtypes :-
    message(123,[info,warn,trace,error,debug]),
    mess_types(X),
    message(124,X),
    read(MT),
    retract(mess_types(_)),
    asserta(mess_types(MT)).

mess(101,info , 'Time for command: ').           % retefoops doit
mess(102,error, 'Invalid Command').             % retefoops do
mess(103,error, 'Initialization Error').        % retefoops initialize
mess(104,trace, 'Rule Firing: ').              % retefoops go
mess(105,trace, 'Conflict Set Delete: ').      % retefoops del_confli...
mess(106,trace, 'Failed to CS Delete: ').      % retefoops del_confli...
mess(107,trace, 'Conflict Set Add: ').         % retefoops add_confli...
mess(108,error, 'Update Fails for: ').         % retefoops update_ws
mess(109,trace, 'Asserting: ').                % retefoops add_ws
mess(110,trace, 'Failing Action Part: ').      % retefoops process
mess(111,error, 'Retract Error, no: ').        % retefoops take
mess(112,debugx, 'Frame error looking for: '). % retefoops find_slot
mess(113,error, 'Frame instance update error: '). % retefoops uptf
mess(114,error, 'No frame to delete: ').       % retefoops del_frame
mess(115,error, 'No instance to delete: ').    % retefoops delf
mess(116,error, 'Unable to delete slot: ').    % retefoops del_slot
mess(117,error, 'Unable to delete facet: ').   % retefoops del_facet
mess(118,trace, 'Rule Fired: ').              % retefoops process
mess(119,error, 'Premature end to run: ').     % retefoops go
mess(120,info, 'Initializing').               % retefoops initialize
mess(121,info, 'Initialization Complete').     % retefoops initialize
mess(122,debugx, 'Conflict Set').             % retefoops select_rule
mess(123,info, 'Legal Message Types: ').      % retefoops set_message
mess(124,info, 'Current Message Types: ').    % retefoops set_message
mess(201,info, 'Rule Rete Network Complete'). % retcomp rete_compil
mess(202,info, 'Rule: ').                    % retcomp rete_comp
mess(203,error, 'Rule Failed to Compile: ').  % retcomp rete_comp

```


F Windows

Windows Demonstration (windemo.pro)

```
% WINDEMO.PRO - demonstrates how to use windows

:- module windemo.

:- public main/0, restart/0.

:- extrn window/2:far, window/3:far.

main:-
  cls,
  go.

restart:-
  halt.

go:-
  create_windows,
  ctr_set(1,1),          % used by list2
  ctr_set(3,1),          % used by dummy
  repeat,
  window(wmain,read,X),
  do(X),
  fail.

create_windows:-
  window(wform, create,
    [type(form),
     coord(8,20,16,53),
     title(' Form not Function '),
     border(white:magenta),contents(white:magenta),
     form([lit(2:5,'First'),
          var(one,2:20,8,''),
          lit(4:5,'Second'),
          var(two,4:20,8,'two'),
          lit(6:5,'Third'),
          var(three,6:20,8,'')
         ]
        )
    ]
  ),
  window(wform2, create,
    [type(form),
     coord(12,24,14,49),
     title(' Form two '),
     border(white:green),contents(white:green),
     form([lit(2:3,'First and Last'),
          var(three,2:20,8,'')
         ]
        )
    ]
  ),
  window(wprompt, create,
    [type(prompt),
     coord(18,10,18,70),
     border(black:green),
```

```

        contents(black:green),
        title(' input ')
    ]
),
window(wmain, create,
    [type(menu),
     coord(15,25,20,40),
     border(blue),
     contents(yellow),
     menu(['new numbers',
          'add numbers',
          'try prompt',
          'try dynamic',
          'try form',
          exit,
          one,two,three,four,five,six,seven
         ]
    )
    ],
),
window(wexit, create,
    [type(display),
     coord(20,40,21,50),
     border(black:red),
     contents(black:red),
     title(' exit ')
    ]
),
window(wdummy, create,
    [type(menu),
     coord(18,32,23,42),
     border(bright:green),
     contents(green:white),
     title(' dummy '),
     menu([return,one,two,three,four,five,six])
    ]
),
window(wdummylog, create,
    [coord(1,1,10,15)]
),
window(wlist1, create,
    [type(display),
     coord(2,2,23,50),
     border(reverse:blue),
     contents(reverse:blue),
     title(' List One ')
    ]
),
window(wlist2, create,
    [type(display),
     coord(2,20,23,78),
     border(yellow),
     contents(blue:yellow),
     title(' List Two ')
    ]
).

do('new numbers'):-
    list1,
    !.
do('add numbers'):-
    list2,
    !.
do('try prompt'):-
    prompt,
    !.
do('try dynamic'):-

```

```

    pop,
    !.
do('try form'):-
    form,
    !.
do(exit):-
    exit.
do(_):-
    dummy.

list1:-
    window(wlist1,open),
    ctr_set(0,1),
    repeat,
        ctr_inc(0,N),
        window(wlist1, write, 'line number is ':N),
        N >= 50,
        window(wlist1, writelist,[nl]),
        window(wlist1, writelist,
            ['You can use home, end, pgup, & pgdn',
             'to examine the contents',
             'use enter to leave the '-wlist1-' window'
            ]
        ),
        window(x, driver),
    window(wlist1, close),
    !.

list2:-
    window(wlist2, open),
    window(wlist2, write, 'adding more numbers'),
    ctr_set(2,1),
    repeat,
        ctr_inc(1,N),
        ctr_inc(2,Test),
        window(wlist2, write, 'adding number':N),
        Test >= 10,
    window(x, driver),
    !.

exit:-
    window(wexit, write, ['Good Bye']),
    window(x, driver),
    cls,
    halt.

dummy:-
    repeat,
        window(wdummy, read, X),
        ctr_inc(3,N),
        window(wdummylog, write, [N:X]),
        X == return,
    window(wdummy, close),
    !.

prompt:-
    repeat,
        window(wprompt, read, [' ',X]),
        window(wdummylog, write, [X]),
        X == ' ',
    window(wprompt, close),
    !.

```

```
pop:-
  window([type(prompt),
    coord(23,2,23,10),
    title(' pop '),
    contents(white:blue)
  ],
  read, [' ',X]
  ),
  window(wdummylog, write, [X]),
  window([type(menu),
    coord(20,2,21,5),
    contents(white:magenta),
    menu([yes,no])
  ],
  read, Y
  ),
  window(wdummylog, write, [Y]),
  !.

form:-
  window(wform, read, _),
  recorded(wform,var(Vname,_,_,Val),_),
  window(wdummylog, write, [Vname=Val]),
  window(wform2, read, _),
  window(wform2, erase),
  fail.
form.
```

Windows (windows.pro)

```
% WINDOWS.PRO - Windowing predicates, written for an old version
% of Arity Prolog. While not operational at this time, they do
% illustrate the power of Prolog for systems type work.

:- module windows.
:- segment(ijseg2).

:- public window/2, window/3.
:- public showcourse/0, hidecourse/0.

:- default(invisible).

:- extrn curtype/2:asm.    % sets the cursor, can be replaced with:

showcourse:-
    curtype(7,8).

hidecourse:-
    curtype(39,40).

%*****
%
% window/3 is the main predicate. The windowing system is organized
% at the top in an object oriented fashion. The window/3 arguments are:
%   arg1 - operation (message)
%   arg2 - window (the object)
%   arg3 - parameters (input or output - either a singleton or a list
%
% The objects can be one of four types of window -
%   display, menu, form, prompt
%
%*****

:- mode window(+,+).

window(W, Op):-
    window(W, Op, []).

:- mode window(+,+,?).

window([H|T], Op, Args):-
    window(temp_w, create, [H|T]),
    window(temp_w, Op, Args),
    window(temp_w, delete),
    !.
window(W, Op, Args):-
    get_type(W, T),
    find_proc(T, Op, Proc),
    doproc(Proc, W, Args),
    !.

% get_type/2 figures out what type of window we have

:- mode get_type(+,-).

get_type(W, X):-
    select_parm(W, [type(X)]),
    !.
get_type(W, window). % W is currently undefined
```

```

% find_proc/3 finds the appropriate procedure for the message
% and window type

:- mode find_proc(+,+,-).

find_proc(T, Op, Proc):-
    find_p(T, Op, Proc),
    !.
find_proc(T, Op, Proc):-
    error([Op, 'is illegal operation for a window of type', T]).

find_p(T, Op, Proc):-
    method(T, Op, Proc),
    !.
find_p(T, Op, Proc):-
    subclass(Super, T),
    !,
    find_p(Super, Op, Proc).

% table of objects

:- mode subclass(+,?).

subclass(window, display).
subclass(window, menu).
subclass(window, form).
subclass(window, prompt).

% table of procedures to use for various operations and types

:- mode method(+,+,-).

method(window, open, open_w).
method(window, close, close_w).
method(window, create, create_w).
method(window, change, change_w).
method(window, driver, driver_w).
method(window, display, display_w).
method(window, delete, delete_w).
method(window, erase, erase_w).
method(display, write, write_d).
method(display, writelist, writelist_d).
method(display, writeline, writeline_d).
method(menu, read, read_m).
method(form, read, read_f).
method(form, display, nop).
method(prompt, read, read_p).

% doproc - a faster way to do a call.

:- mode doproc(+,+,+).

doproc(create_w,W,A):-
    !,
    create_w(W,A).
doproc(open_w,W,A):-
    !,
    open_w(W,A).
doproc(close_w,W,A):-
    !,
    close_w(W,A).
doproc(delete_w,W,A):-

```

```

    !,
    delete_w(W,A) .
doproc(display_w,W,A):-
    !,
    display_w(W,A) .
doproc(erase_w,W,A):-
    !,
    erase_w(W,A) .
doproc(change_w,W,A):-
    !,
    change_w(W,A) .
doproc(write_d,W,A):-
    !,
    write_d(W,A) .
doproc(writelst_d,W,A):-
    !,
    writelst_d(W,A) .
doproc(writeline_d,W,A):-
    !,
    writeline_d(W,A) .
doproc(read_m,W,A):-
    !,
    read_m(W,A) .
doproc(read_f,W,A):-
    !,
    read_f(W,A) .
doproc(read_p,W,A):-
    !,
    read_p(W,A) .
doproc(driver_w,W,A):-
    !,
    driver_w(W,A) .
doproc(nop,_,_):- !.
doproc(X,W,A):-
    error(['No window method ',X,' defined.']).

%*****
%
% methods for the super class "window".  these are used by default
% if they are not redefined for the subclass.
%
%     create_w - create new window from specs
%     open_w   - open a window for use
%     close_w  - remove the window contents and viewport
%     change_w - changes a windows definition
%     display_w - display a portion of contents in window
%     driver_w - gives control to user to view other windows
%
%*****

%-----
%
% create_w/2 records a new window definition
%
%-----

:- mode create_w(+,?) .

create_w(W,L):-
    make_window(W,L),
    !.

%-----
%
% open_w/1 calls make_viewport which opens up a viewport for the window
%
```

```

%-----
:- mode open_w(+,?).

open_w(W, []):-
    exists_window(W),
    make_viewport(W),
    !.

%-----
%
% close_w/1 closes a viewport on the active list
%
%-----

:- mode close_w(+,?).

close_w(W, []):-
    del_viewport(W),
    !.

%-----
%
% delete_w removes both viewport and dataarea
%
%-----

:- mode delete_w(+,?).

delete_w(W, []):-
    del_viewport(W),
    del_dataarea(W),
    del_stat(W),
    del_image(W),
    del_window(W),
    !.

%-----
%
% erase_w removes the viewport and dataarea, but preserves
% window definition
%
%-----

:- mode erase_w(+,?).

erase_w(W, _):-
    del_viewport(W),
    del_dataarea(W),
    del_stat(W),
    del_image(W),
    !.

%-----
%
% change_w/2 changes a windows position on the viewport
%
%-----

:- mode change_w(+,?).

change_w(W, L):-
    recorded_w(windef, wd(W, Lold), _),
    merge_wl(L, Lold, Lnew),

```

```

w_clrb(W),
recorded_w(active,AL,_),
remove(W,AL,NL),
redraw(NL,W),
recorded_w(active,AL,DBRef), erase(DBRef),
recorda(active,NL,_),
window(W,create,Lnew),
window(W,open),
!.

merge_wl([],Lnew,Lnew):- !.
merge_wl([Hn|Tn], Lold, Lnew):-
    rep_we(Hn, Lold, Temp),
    !,
    merge_wl(Tn, Temp, Lnew).

rep_we(X, [], [X]):- !.
rep_we(X, [H|T], [X|T]):-
    functor(X,F,A),
    functor(H,F,A),
    !.
rep_we(X, [H|T], [H|T2]):-
    !,
    rep_we(X, T, T2).

%-----
%
% display_w writes a page of a window from a given starting point
%
%-----

:- mode display_w(+,?).

display_w(W, [_ , 0]):- !.
display_w(W, [Line, NN]):- % add NN lines to the top
    NN < 0,
    N is -NN,
    select_parm(W, [coord(R1, C1, R2, C2)]),
    RLL is R1 + N - 1,
    (RLL =< R2,
     RL = RLL;
     RL = R2
    ),
    display_viewport(W, Line, R1, RL, C1),
    !.
display_w(W, [Line, NN]):- % add NN lines to the bottom
    NN > 0, % note Line is line number at top of window
    select_parm(W, [coord(R1, C1, R2, C2)]),
    RFF is R2 - NN + 1,
    (RFF =< R1,
     RF = R1;
     RF = RFF
    ),
    Offset is RF - R1, % if first line to be displayed
    Lineoff is Line + Offset, % is mid viewport somewhere
    display_viewport(W, Lineoff, RF, R2, C1),
    !.
display_w(W, Line):-
    select_parm(W, [coord(R1, C1, R2, C2)]),
    display_viewport(W, Line, R1, R2, C1),
    !.

%-----
%
```

```

% driver_w turns control over to the user for manipulating the
% current window.
%
%-----

driver_w(,_):-
  repeat,
  recorded_w(active,[W|_],_),
  select_parm(W,[coord(R1,C1,_,_)]),
  tmove(R1,C1),
  keyb(A,S),
  w_exec(S,Flag,W),
  Flag == end,
  !.

:- mode w_exec(+,-,+).

w_exec(71,xxx,W):-          % home
  scroll_window(W,top),
  !.
w_exec(79,xxx,W):-          % end
  scroll_window(W,bottom),
  !.
w_exec(81,xxx,W):-          % pgdn
  select_parm(W,[height(H)]),
  HH is H - 1,
  scroll_window(W,HH),
  !.
w_exec(73,xxx,W):-          % pgup
  select_parm(W,[height(H)]),
  HH is - H + 1,
  scroll_window(W,HH),
  !.
w_exec(72,xxx,W):-          % up arrow
  scroll_window(W,-1),
  !.
w_exec(80,xxx,W):-          % down arrow
  scroll_window(W,1),
  !.
w_exec(59,xxx,W):-          % f1 change windows
  recorded_w(active,List,_),
  last_item(List,NewW),
  window(NewW,open),
  !.
w_exec(28,end,_):- !.      % enter - leave the driver
w_exec(_,xxx,_):- !.

%*****
%
% methods for subclass display
%
%   write_d - write to the window
%   writelist_d - write a list of terms to the window
%
%*****

%-----
%
% write_d
% write to the window. The term can be a simple term, or a list of terms
% which make up a line. See w_wlin for other allowed constructs.
%
%-----

:- mode write_d(+,?).

```

```

write_d(W, Term):-
  scroll_window(W, bottom),
  select_stat(W, curnum, L1, NL),
  select_parm(W, [coord(R1, C1, R2, C2), attr(CC)]),
  Row is NL - L1 + 1 + R1,
  add_data(W, Term),
  write_viewport(W, Term, CC, L1, NL, Row, R2, C1, C2),
  !.

%-----
%
% writelist_d
% write multiple lines to the window, using write_d
%
%-----

:- mode writelist_d(+,?).

writelist_d(W, []):-!.
writelist_d(W, [H|T]):-
  w_writ(W, H),
  !,
  writelist_d(W, T).

w_writ(W, nl):-
  write_d(W, '').
w_writ(W, H):-
  write_d(W, H).

%-----
%
% writeline_d
% write a list of terms on a line, without the list format
%
%-----

:- mode writeline_d(+,?).

writeline_d(W, L):-
  make_line(L, $$, S),
  write_d(W, S),
  !.

make_line([], S, S).
make_line([H|T], Temp, S):-
  string_term(HS, H),
  concat(Temp, HS, Temp2),
  make_line(T, Temp2, S).

%*****
%
% methods for subclass menu
%
%   read_m - read a term from the menu
%
%*****

%-----
%
% read_m
% w_menu - returns a menu choice from a menu window, the window may
% be dynamically built, using the first clause
%
```

```

%-----
:- mode read_m(+,?).

read_m(W,X):-
  init_menu_dataarea(W),
  window(W,open),
  menu_select(W,X),
  !.
% (string_term(X,XX); atom_string(XX,X)), !.

%*****
%
% methods for subclass form
%
%   read_f - read the fields in the form
%
%*****
%-----
%
% read_f - updates the contents of the form window
%
%-----

:- mode read_f(+,?).

read_f(W,[]):-
  init_form_dataarea(W),
  window(W,open),
  clear_viewport(W),
  display_form(W),
  select_parm(W,[coord(_,C1,R2,_)]),
  RR is R2 + 1,
  tmove(RR,C1),
  write(' F9 to enter '),
  fill_form(W),
  read_form(W),
  !.

%*****
%
% methods for subclass prompt
%
%*****
%-----
%
% read_p read the prompt
%
%-----

:- mode read_p(+,?).

read_p(W,X):-
  var(X),
  window(W,open),
  fill_prompt(W,Z),
  !,
  Z = X.
read_p(W,X):-
  atomic(X),
  window(W,open),
  fill_prompt(W,Z),
  !,
  Z == X.                                     % ok if X an atom and Z a string

```

```

read_p(W, [Def, X]) :-
    write_d(W, Def),
    fill_prompt(W, Z),
    !,
    Z = X.

%*****
%
% Basic window data manipulation routines - a window is composed
% of a viewport and dataarea
%
%*****

%-----
%
% make a new window
%
%-----

:- mode make_window(+, ?).

make_window(W, Def) :-
    (recorded_w(windef, wd(W, _), DBRef),
     erase(DBRef);
     true
    ),
    recorda(windef, wd(W, Def), _),
    !.

%-----
%
% del_window removes a window definition
%
%-----

:- mode del_window(+).

del_window(W) :-
    recorded_w(windef, wd(W, _), DBRef),
    erase(DBRef),
    !.
del_window(_).

%-----
%
% exists_window checks for existence of a window definition
%
%-----

:- mode exists_window(+).

exists_window(W) :-
    recorded_w(windef, wd(W, _), _),
    !.
exists_window(W) :-
    error(['No window definition for ', W]),
    fail.

%-----
%
% select_parm extracts various parameters from a window definition.
% The RequestList is a list of structures with keyword functors
% and variable arguments, which are bound by w_attr. w_attr
% also contains the defaults and computed parameters (ie height)

```

```

%
%-----

:- mode select_parm(+,+).

select_parm(W,RequestList):-
    recorded_w(windef,wd(W,AttrList),_),
    fullfill(RequestList, AttrList),
    !.

:- mode fullfill(+,+).

fullfill([],_):- !.
fullfill([Req|T], AttrList):-
    w_attr(Req, AttrList),
    !,
    fullfill(T, AttrList).

:- mode w_attr(+,+).

w_attr(height(H), AttrList):-
    w_attr(coord(R1,_,R2,_),AttrList),
    H is R2 - R1 + 1,
    !.
w_attr(width(W), AttrList):-
    w_attr(coord(_,C1,_,C2),AttrList),
    W is C2 - C1 + 1,
    !.
w_attr(attr(A), AttrList):-
    w_attr(contents(Color),AttrList),
    attr(Color,A),
    !.
w_attr(border_attr(A), AttrList):-
    w_attr(border(Color),AttrList),
    attr(Color,A),
    !.
w_attr(A, AttrList):-
    member(A, AttrList),
    !.
w_attr(coord(1,1,23,78),_).           % default values
w_attr(title(''),_).
w_attr(border(white),_).
w_attr(contents(white),_).
w_attr(type(display),_).

%
%-----
%
% update_parm provides the facility to update and window parameters.
%
%-----

:- mode update_parm(+,?).

update_parm(W,UpdateList):-
    recorded_w(windef,wd(W,AttrList),DBRef), erase(DBRef),
    modify(UpdateList, AttrList, NewList),
    recorda(windef,wd(W,NewList),_),
    !.

modify([],L,L):-!.
modify([Req|T], AttrList, NewList):-
    functor(Req,F,A),
    mod(F, Req, AttrList, [], NewL),
    !,

```

```

modify(T, NewL, NewList).

mod(_,_, [], L, L).
mod(F, A, [OldA|AL], Temp, NewL) :-
    functor(OldA, F, _),
    append([A|Temp], AL, NewL),
    !.
mod(F, A, [OldA|AL], Temp, NewL) :-
    mod(F, A, AL, [OldA|Temp], NewL).

%-----
%
% select_stat is used to get the status of a window
%
%-----

:- mode select_stat(+,+,?).

select_stat(W, curline, L) :-
    recorded_w(curline, cl(W, L, _), _),
    !.
select_stat(W, numlines, N) :-
    recorded_w(curline, cl(W, _, N), _).

:- mode select_stat(+,+,?,?).

select_stat(W, curnum, L, N) :-
    recorded_w(curline, cl(W, L, N), _).

%-----
%
% update_stat is used to change the active status of a window
%
%-----

:- mode update_stat(+,+,?).

update_stat(W, curline, L) :-
    (recorded_w(curline, cl(W, _, NL), DBRef),
     erase(DBRef);
     NL = 0
    ),
    !,
    recorda(curline, cl(W, L, NL), _).
update_stat(W, numlines, N) :-
    (recorded_w(curline, cl(W, L, _), DBRef),
     erase(DBRef);
     L = 1
    ),
    !,
    recorda(curline, cl(W, L, N), _).

:- mode update_stat(+,+,?,?).

update_stat(W, curnum, L, N) :-
    (recorded_w(curline, cl(W, _, _), DBRef),
     erase(DBRef);
     true
    ),
    !,
    recorda(curline, cl(W, L, N), _).

```

```

%-----
%
% del_stat removes the windows status information
%
%-----

:- mode del_stat(+).

del_stat(W):-
    recorded_w(curline,cl(W,_,_),DBRef),
    erase(DBRef),
    !.
del_stat(_).

%-----
%
% select_content will repeatedly give the next record from
% a given starting point, and its position
%
%-----

:- mode select_content(+,?,?,?).

select_content(W,Start,Count,X):-
    ctr_set(20,0),
    recorded_w(W,X,_),
    ctr_inc(20,Count),
    Count >= Start.

%-----
%
% add to a windows data area
%
%-----

:- mode add_data(+,?).

add_data(W, Term):-
    (string(Term),
     S = Term;
     string_term(S,Term)
    ),
    recordz(W, S, _),
    !.

%-----
%
% make_viewport initializes a viewport.  If it is already the head of the
% active list, do nothing.  If it is not on the active list, get it
% and put it on.  Otherwise move it to the head from where it is now.
%
%-----

:- mode make_viewport(+).

make_viewport(W):-
    recorded_w(active,[W|T],_),           % If its on the top, clear the decks
    del_image(W),
    !.
make_viewport(W):-
    w_inact(W),                             % make sure active and curline exist
    w_nocur(W),                             % and go to next clause
    fail.
make_viewport(W):-
    recorded_w(active,[H|T],_),           % If its on the list somewhere

```

```

save_image(H),
split(W,[H|T],L1,L2),
w_chkover(W,L1,_),
append([W|L1],L2,NL),
recorded_w(active,_,DBRef), erase(DBRef),
recorda(active,NL,_),
!.
make_viewport(W):-
recorded_w(active,L,DBRef), erase(DBRef),
recorda(active,[W|L],_),
w_ini(W),
!.
make_viewport(W):-
error(['Initializing viewport',W]).

w_inact(W):-
recorded_w(active,_,_),
!.
w_inact(W):-
recorda(active,[],_).

w_nocur(W):-
select_stat(W,curnum,_,_),
!.
w_nocur(W):-
update_stat(W,curnum,1,0).

w_ini(W):-
w_box(W),
clear_viewport(W),
window(W,display,1),           % display from line 1
set_arrows(W),
!.

w_chkover(W,[],no).
w_chkover(W,[H|T],Stat):-
w_nooverlap(W,H),
w_chkover(W,T,Stat).
w_chkover(W,_,yes):-
restore_image(W),
!.

:- mode w_nooverlap(+,+).

w_nooverlap(Wa,Wb):-
select_parm(Wa,[coord(R1a,C1a,R2a,C2a)]),
select_parm(Wb,[coord(R1b,C1b,R2b,C2b)]),
(R1a > R2b + 2;
 R2a < R1b - 2;
 C1a > C2b + 2;
 C2a < C1b - 2
),
!.

%-----
%
% save the screen image
%
%-----

:- mode save_image(+).

```

```

save_image(W):-
  del_image(W),
  select_parm(W,[coord(R1,C1,R2,C2)]),
  RR1 is R1 - 1, CC1 is C1 - 1,
  RR2 is R2 + 1, CC2 is C2 + 1,
  region_ca((RR1,CC1),(RR2,CC2),SA),
  recorda(image,W-SA,_),
  !.

%-----
%
% restore the screen image
%
%-----

:- mode restore_image(+).

restore_image(W):-
  select_parm(W,[coord(R1,C1,R2,C2)]),
  RR1 is R1 - 1, CC1 is C1 - 1,
  RR2 is R2 + 1, CC2 is C2 + 1,
  recorded_w(image,W-SA,Ref),
  region_ca((RR1,CC1),(RR2,CC2),SA),
  !.

%-----
%
% delete an image
%
%-----

:- mode del_image(+).

del_image(W):-
  recorded_w(image,W_,R),
  erase(R),
  !.
del_image(W).

%-----
%
% del_dataarea removes the contents of the window
%
%-----

:- mode del_dataarea(+).

del_dataarea(W):-
  eraseall(W),
  update_stat(W,curnum,1,0),
  !.

%-----
%
% clear_viewport clears the screen
%
%-----

:- mode clear_viewport(+).

clear_viewport(W):-
  select_parm(W,[coord(R1,C1,R2,C2),attr(A)]),
  tmove(R2,C1),
  wca(1,`,A),

```

```

    tscroll(0, (R1,C1), (R2,C2)),
    !.
clear_viewport(_).

%-----
%
% del_viewport removes a viewport updating active lists and overlays
%
%-----

:- mode del_viewport(+).

del_viewport(W):-
    recorded_w(active,AList,_),
    member(W,AList),                %fail and go away if nothing to delete
    AList = [H|_],
    (W == H; save_image(H)),        % in case closing a window other
    w_clrb(W),                       % than the uppermost
    remove(W,AList,Newl),
    del_image(W),
    redraw(Newl,W),
    recorded_w(active,AList,DBRef), erase(DBRef),
    recorda(active,Newl,_),
    !.
del_viewport(_).                    % always succeed

redraw(Newl,W):-                    % from back to front, redraw
    reverse(Newl,Backwards),        % affected windows
    redr(Backwards,W,[]),
    !.

redr([],_,_):- !.
redr([H|T], W, Redrawn):-
    w_nooverlap(H,W),
    w_chkover(H,Redrawn,Stat),
    (Stat == yes,
     Red = [H|Redrawn];
     Red = Redrawn
    ),
    !,
    redr(T,W,Red).
redr([H|T], W, Redrawn):-
    restore_image(H),
    !,
    redr(T, W, [H|Redrawn]).

% remove the window from the viewport

:- mode w_clrb(+).

w_clrb(W):-
    select_parm(W, [coord(R1, C1, R2, C2)]),
    RR1 is R1 - 1, CC1 is C1 - 1,
    RR2 is R2 + 1, CC2 is C2 + 1,
    tmove(RR2,CC1),wca(1,`,7),
    tscroll(0, (RR1, CC1), (RR2, CC2)),
    !.
w_clrb(W):- !.

%-----
%
% writes a term to the specified line in the viewport
%
%-----

```

```

:- mode write_viewport(+,+,+,+,+,+,+,+,+).

write_viewport(W, Term, CC, L1, NL, Row, R2, C1, C2):-
  (Row =< R2,
   R = Row,
   L2 is L1,
   !;
   scroll_viewport(W, 1),
   L2 is L1 + 1, R = R2
  ),
  Width is C2 - C1 + 1,
  (string(Term),
   S = Term;
   string_term(S,Term)
  ),
  w_wline(S, R, C1, CC, Width),
  NNL is NL + 1,
  update_stat(W,curnum,L2,NNL),
  !.

:- mode w_wline(+,+,+,+,+).

w_wline(Term, R, C1, CC, Width):-
  tmove(R,C1),
  wa(Width,CC),
  (string_length(Term,L),
   L =< Width,
   Term = S;
   substring(Term,0,Width,S)
  ),
  write(S).

%-----
%
% display_viewport writes lines from R1 to R2 starting at line L
% from the dataarea to the viewport
%
%-----

:- mode display_viewport(+,+,+,+,+).

display_viewport(W,Line,R1,R2,C1):-
  key(W,Key),
  nth_ref(W,Line,Ref),
  select_parm(W,[width(Wid), attr(A)]),
  RL is R2 + 1,
  w_disp(Ref, Key, R1, RL, C1, A, Wid),
  !.
display_viewport(_,_,_,_,_).          % succeed if no key yet

:- mode w_disp(+,+,+,+,+,+,+).

w_disp(Ref, Ref, _, _, _, _):- !.
w_disp(_,_ , Row, Row, _, _):- !.
w_disp(Ref, Sref, Row, RL, C1, CC, Width):-
  instance(Ref, Term),
  w_wline(Term, Row, C1, CC, Width),
  Row2 is Row + 1,
  nref(Ref, Nref),
  !,
  w_disp(Nref, Sref, Row2, RL, C1, CC, Width).

%-----

```

```

%
% the box, only one choice, double line
%
%-----

:- mode w_box(+).

w_box(W):-
  select_parm(W, [coord(R1, C1, R2, C2), title(T), border(C)]),
  box(R1, C1, R2, C2, C),
  Rt is R1 - 1,
  Ct is C1 + 3,
  tmove(Rt,Ct),
  write(T),
  !.

:- mode box(+,+,+,+,+).

box(R1, C1, R2, C2, C):-
  R0 is R1 - 1,
  C0 is C1 - 1,
  R3 is R2 + 1,
  C3 is C2 + 1,
  Width is C2 - C1 + 1,
  Height is R2 - R1 + 1,
  attr(C,CC),
  left_side(R1, C0, CC, R2),
  top(R0, C0, C1, C3, Width, CC),
  right_side(R1, C3, CC, R2),
  bottom(R3, C0, C1, C3, Width, CC),
  !.

:- mode left_side(+,+,+,+).

left_side(R1, C0, CC, R2):-
  w_vert(R1, C0, 186, CC, R2).

:- mode top(+,+,+,+,+).

top(R0, C0, C1, C3, Width, CC):-
  tmove(R0, C0),
  wca(1, 201, CC),
  tmove(R0, C1),
  wca(Width, 205, CC),
  tmove(R0, C3),
  wca(1, 187, CC).

:- mode right_side(+,+,+,+).

right_side(R1, C3, CC, R2):-
  w_vert(R1, C3, 186, CC, R2).

:- mode bottom(+,+,+,+,+).

bottom(R3, C0, C1, C3, Width, CC):-
  tmove(R3, C0),
  wca(1,200,CC),
  tmove(R3, C1),
  wca(Width, 205, CC),
  tmove(R3,C3),
  wca(1, 188, CC).

```

```

:- mode w_vert(+,+,+,+,+).

w_vert(R1, C1, Char, Color, R2):-
  ctr_set(0,R1),
  repeat,
  ctr_inc(0,R),
  tmove(R, C1),
  wca(1, Char, Color),
  R >= R2,
  !.

%-----
%
% scroll the viewport
%
%-----

:- mode scroll_viewport(+,?).

scroll_viewport(W, N):-
  select_parm(W, [coord(R1, C1, R2, C2)]),
  Height is R2 - R1 + 1,
  Heightm is -Height,
  (N > Heightm,
   N < Height,
   NN = N;
   NN = 0
  ),
  tscroll(NN, (R1,C1), (R2,C2)).

%-----
%
% scroll the window
%
%-----

:- mode scroll_window(+,+).

scroll_window(W, top):-
  window(W,open),
  select_stat(W,curline,1),          % already at the top
  set_arrows(W),
  !.
scroll_window(W, top):-
  select_stat(W,curline,L),
  S is 1 - L,
  scroll_window(W,S),
  !.
scroll_window(W, bottom):-
  window(W,open),
  select_parm(W, [coord(R1, C1, R2, C2)]),
  Height is R2 - R1 + 1,
  select_stat(W, curnum, L, NL),
  NL < L + Height, % already at the bottom
  set_arrows(W),
  !.
scroll_window(W, bottom):-
  select_parm(W, [coord(R1, C1, R2, C2)]),
  Height is R2 - R1 + 1,
  select_stat(W, curnum, L, NL),
  Last is L + Height,
  S is NL + 1 - Last,
  scroll_window(W, S),
  !.
scroll_window(W, N):-
  window(W,open),

```

```

select_stat(W, curnum, Line, NL),
select_parm(W, [height(H)]),
H < NL, % if it fits on one frame, no scroll
MaxLine is NL - H + 1, % biggest line # allowed at first row
Newline is Line + N,
real_nl(in(Line,MaxLine,Newline,N), out(Newl,NN)),
set_arrow(W,MaxLine,Newl),
update_stat(W, curnum, Newl, NL),
scroll_viewport(W, NN),
window(W, display, [Newl, NN]),
!.
scroll_window(_,_).

%-----
%
% set_arrows puts the little arrows at the top and bottom indicating
% that there is more to be seen.
%
%-----

:- mode set_arrows(+).

set_arrows(W):-
  select_stat(W, curnum, Line, NL),
  select_parm(W, [height(H)]),
  Max is NL - H + 1,
  set_arrow(W, Max, Line),
  !.

set_arrow(W,Max,New):-
  select_parm(W, [coord(R1,C1,R2,C2)]),
  RR1 is R1 - 1,
  RR2 is R2 + 1,
  setar(Max,New,up(RR1:C1),down(RR2:C1)),
  !.

setar(_, 1, up(R:C), _):-
  tmove(R,C),
  put(`í`),
  fail.
setar(Max, Max, _, down(R:C)):-
  tmove(R,C),
  put(`í`),
  fail.
setar(_, New, up(R:C), _):-
  New > 1,
  tmove(R,C),
  put(`□`),
  fail.
setar(Max, New, _, down(R:C)):-
  New < Max,
  tmove(R,C),
  put(`□`),
  fail.
setar(_,_,_,_).

real_nl(in(Line,MaxLine,Newline,N), out(Newline,N)):-
  Newline > 0,
  Newline =< MaxLine,
  !.
real_nl(in(Line,MaxLine,Newline,N), out(1,NN)):-
  Newline =< 0,
  Newline =< MaxLine,
  NN is 1 - Line,
  Line =\= 1, % fail if already at first line

```

```

!.
real_nl(in(Line,MaxLine,Newline,N), out(MaxLine,NN)):-
  Newline > 0,
  Newline > MaxLine,
  NN is MaxLine - Line,
  Line =\= MaxLine, % fail if already at last line
!.

%-----
%
% predicates to define and read maps, aka forms.
% they expect an initial definition of the map fields
% in a window spec as follows:
%
%      form([lit(Row:Col,Literal),
%           var(FieldName,Row:Col,Length,InitValue),
%           ...])
%
% This is converted to window records of the form
%
%      recordz(W,lit(..),_). etc
%
% It is sometimes easier for the application to build this directly.
%-----
%-----
%
% copy the window definition specs to the dataarea
%-----

:- mode init_form_dataarea(+).

init_form_dataarea(F):-
  recorded_w(F,_,_),
  !.
init_form_dataarea(F):-
  select_parm(F,[form(List)]),
  init_map(F,List),
  !.

:- mode init_map(+,?).

init_map(W,[]):-!.
init_map(W,[H|T]):-
  recordz(W,H,_),
  !, init_map(W, T).

%-----
%
% put the form data on the viewport
%-----

:- mode display_form(+).

display_form(S):-
  select_parm(S,[coord(R0,C0,_,_),attr(At)]),
  recorded_w(S,Field,_),
  write_field(R0:C0,At,Field),
  fail.
display_form(S):-true.

write_field(R0:C0, At, lit(R:C,Lit)):-

```

```

RR is R0 + R, CC is C0 + C,
tmove(RR,CC),
write(Lit),
!.
write_field(R0:C0, At, var(_,R:C,Length,Val)):-
  rev_attr(At, Rat),
  RR is R0 + R, CC is C0 + C,
  tmove(RR,CC),
  wa(Length,Rat),
  write(Val),
  !.

%-----
%
% read the data from the viewport and store in the dataarea
%
%-----

:- mode read_form(+).

read_form(S):-
  select_parm(S, [coord(R0,C0,_,_) ]),
  recorded_w(S,var(Name,R:C,Length,_),Ref),
  RR is R0 + R, CC is C0 + C,
  C2 is CC + Length - 1,
  region_c((RR,CC), (RR,C2), Str),
  strcnv(Str,Val),
  replace(Ref,var(Name,R:C,Length,Val)),
  fail.
read_form(S).

strcnv(S,V):-
  strip_leading(S,S1),
  strip_trailing(S1,V),
  !.

%-----
%
% capture keystrokes and drive form data entry
%
%-----

:- mode fill_form(+).

fill_form(S):-
  build_field_list(S),
  recorded_w(field_list,S-[R:C:C2|T],_),
  tmove(R,C),
  set_flag(current_field,R:C:C2),
  get_keystrokes(S,[R:C:C2|T]).

build_field_list(S):-
  (recorded_w(field_list,S-L,Ref),erase(Ref); true),
  recorda(field_list,S-[ ],_),
  bfl(S),
  !.

bfl(S):-
  select_parm(S, [coord(R0,C0,_,_) ]),
  recorded_w(S,var(_,R:C,Length,_),_),
  RR is R0 + R, CC is C0 + C,
  C2 is CC + Length - 1,
  add_field(S,RR:CC:C2),

```

```

fail.
bfl(S):-
    recorded_w(field_list,S-L,Ref), % If there is only one, make
    length(L,N), N == 1,           % a dummy second one so next_item
    append(L,L,L2),                % has something to find
    erase(Ref),
    recorda(field_list,S-L,Ref),
    !.
bfl(_).

add_field(S,F):-
    recorded_w(field_list,S-L,Ref),
    append(L,[F],L2),
    erase(Ref),
    recorda(field_list,S-L2,_),
    !.

get_keystrokes(W,List):-
    select_parm(W,[attr(At)]),
    rev_attr(At, Rat),
    repeat,
        get_flag(current_field,F),
        keyb(A,S),
        put_viewport(A:S,F,List,Rat),
    !. % user ended input

put_viewport(A:67, F, FList, Rat). % f9
put_viewport(A:77, F, FList, Rat):- % rt arrow
    curse_inc(F, FList),
    !,
    fail.
put_viewport(A:75, F, FList, Rat):- % left arrow
    curse_dec(F, FList),
    !,
    fail.
put_viewport(A:72, F, FList, Rat):- % up arrow
    prev_item(F, FList, R:C:C2),
    set_flag(current_field, R:C:C2),
    tmove(R,C),
    !,
    fail.
put_viewport(A:80, F, FList, Rat):- % down arrow
    next_item(F, FList, R:C:C2),
    set_flag(current_field, R:C:C2),
    tmove(R,C),
    !,
    fail.
put_viewport(A:28, F, FList, Rat):- % enter
    next_item(F, FList, R:C:C2),
    set_flag(current_field, R:C:C2),
    tmove(R,C),
    !,
    fail.
put_viewport(A:14, F, FList, Rat):- % back space
    curse_dec(F, FList),
    wca(1, ` , Rat),
    !,
    fail.
put_viewport(A:_, F, FList, Rat):- % letter
    wca(1, A, Rat),
    curse_inc(F, FList),
    !,
    fail.

```

```

curse_inc(Rx:Cx:C2x, FList):-
    tget(R,C),
    C < C2x,
    CC is C + 1,
    tmove(R,CC),
    !.
curse_inc(F, FList):-
    next_item(F, FList, R:C:C2),
    set_flag(current_field, R:C:C2),
    tmove(R,C),
    !.

curse_dec(Rx:Cx:C2x, FList):-
    tget(R,C),
    C > Cx,
    CC is C - 1,
    tmove(R,CC),
    !.
curse_dec(F, FList):-
    prev_item(F, FList, R:C:C2),
    set_flag(current_field, R:C:C2),
    tmove(R,C),
    !.

%-----
%
% write the menu dataarea from the window definition
%
%-----

:- mode init_menu_dataarea(+).

init_menu_dataarea(W):-
    recorded_w(W,_,_),
    !.
init_menu_dataarea(W):-
    m_init(W).

m_init(W):-
    select_parm(W, [menu(ItemList)]),
    m_create(W, ItemList, Nitems, 0),
    update_stat(W, curnum, 1, Nitems),
    !.

m_create(_, [], Nitems, Nitems):- !.
m_create(W, [Item|Rest], Nitems, X):-
    add_data(W, Item),
    XX is X + 1,
    !,
    m_create(W, Rest, Nitems, XX).

%-----
%
% select an item from the menu
%
%-----

:- mode menu_select(+,?).

menu_select(W, X):-
    select_parm(W, [coord(R1, C1, R2, _),
    width(L),
    attr(A)]),

```

```

tmove(R1,C1),
revideo(L,A),
repeat,
  keyb(_,S),
  m_cur(S,Z,w(W,R1,R2,C1,L,A)), % will fail until Z has a value
!,
Z = X. % might fail if X had a value

:- mode m_cur(+,-,+).

m_cur(80,_,w(W,R1,R2,C1,L,A)):- % down arrow
  tget(R,_),
  R < R2,
  normvideo(L,A),
  RR is R + 1,
  tmove(RR,C1),
  revideo(L,A),
  !,
  fail.
m_cur(80,_,w(W,R1,R2,C1,L,A)):- % down arrow at bottom
  tget(R,_),
  R >= R2,
  normvideo(L,A),
  scroll_window(W,1),
  tmove(R2,C1),
  revideo(L,A),
  !,
  fail.
m_cur(72,_,w(W,R1,R2,C1,L,A)):- % up arrow
  tget(R,_),
  R > R1,
  normvideo(L,A),
  RR is R - 1,
  tmove(RR,C1),
  revideo(L,A),
  !,
  fail.
m_cur(72,_,w(W,R1,R2,C1,L,A)):- % up arrow at top
  tget(R,_),
  R =< R1,
  normvideo(L,A),
  scroll_window(W,-1),
  tmove(R1,C1),
  revideo(L,A),
  !,
  fail.
m_cur(71,_,w(W,R1,R2,C1,L,A)):- % home
  normvideo(L,A),
  scroll_window(W,top),
  tmove(R1,C1),
  revideo(L,A),
  !,
  fail.
m_cur(79,_,w(W,R1,R2,C1,L,A)):- % end
  normvideo(L,A),
  scroll_window(W,bottom),
  tmove(R2,C1),
  revideo(L,A),
  !,
  fail.
m_cur(28,X,w(W,R1,R2,C1,L,A)):- % enter
  tget(R,_),
  select_stat(W,curline,Line),
  Nth is Line + R - R1,
  nth_ref(W, Nth, Ref),
  instance(Ref,X),
  normvideo(L,A),

```

```

!.

revideo(L,A):-
  rev_attr(A,RevAt),
  wa(L,RevAt).

normvideo(L,A):-
  wa(L,A).

%-----
%
% read the prompt
%
%-----

:- mode fill_prompt(+,?).

fill_prompt(W,Z):-
  select_parm(W,[coord(R1,C1,_,_), width(L), contents(Color)]),
  attr(Color,At),
  repeat,
  C2 is C1 + L - 1,
  tmove(R1,C1),
  lined(Y,R1:C1:C2,At),
  strip_leading(Y,Y1),
  strip_trailing(Y1,X),
  !,
  Z = X. % might fail if checking X value

% lined - a line editor, returns a string

% :- mode lined(?,+,+).

lined(NewS,R:C1:C2,At):-
  L is C2 - C1 + 1,
  tmove(R,C1),
  wa(L,At),
  repeat,
  keyb(A,S),
  modify(A:S,R:C1:C2,EndFlag,At),
  EndFlag == end,
  region_c((R,C1),(R,C2),NewS),
  !.

%:- mode modify(+,+,?,+).

modify(_:28,R:C1:C2,end,At):- !. % CR - end edit
modify(_:1,R:C1:C2,x,At):- % Esc - erase line
  tscroll(0,(R,C1),(R,C2)),
  tmove(R,C1),
  !.
modify(_:77,R:C1:C2,x,At):- % cursor right
  tget(R,C),
  (C < C2,
   CC is C + 1;
   CC = C2
  ),
  tmove(R,CC),
  !.
modify(_:75,R:C1:C2,x,At):- % cursor left
  tget(R,C),
  (C > C1,
   CC is C - 1;

```

```

        CC = C1
    ),
    tmove(R,CC),
    !.
modify(_:83,R:C1:C2,x,At):-                % Del - delete character
    tget(R,C),
    C < C2,
    CC is C + 1,
    hidecourse,
    region_ca((R,CC),(R,C2),S),
    list_text([32,At],LastChar),
    concat(S,LastChar,St),
    region_ca((R,C),(R,C2),St),
    tmove(R,C),
    showcourse,
    !.
modify(_:83,R:C1:C2,x,At):-                % Del - last space
    wca(1,`,At),
    !.
modify(_:14,R:C1:C2,x,At):-                % BS - move everything to the left
    tget(R,C),
    C > C1,
    CC is C - 1,
    hidecourse,
    region_ca((R,C),(R,C2),S),
    list_text([32,At],LastChar),
    concat(S,LastChar,St),
    region_ca((R,CC),(R,C2),St),
    tmove(R,CC),
    showcourse,
    !.
modify(_:14,_,_,_):- !.                    % BS - first space
modify(A:_,R:C1:C2,x,At):-                % any other character, insert
    tget(R,C),
    C < C2, CC is C + 1,
    hidecourse,
    C2a is C2 - 1,
    region_ca((R,C),(R,C2a),S),
    tmove(R,C),
    wca(1,A,At),
    region_ca((R,CC),(R,C2),S),
    tmove(R,CC),
    showcourse,
    !.
modify(A:_,_,_,At):-                      % last space
    wca(1,A,At).

%-----
%
% debug dumps all the current window data
%
%-----

debug:-
    write('window dump'), nl,
    debug_active,
    debug_windefs,
    !.

debug_active:-
    recorded_w(active,L,_),
    write(active:L), nl,
    !.
debug_active:-
    write('no active list'), nl.

```

```

debug_windefs:-
    recorded_w(windef,wd(W,A),_),
    write(W:A), nl,
    debug_curline(W),
    debug_data(W),
    fail.
debug_windefs:-
    write('no more windefs'), nl.

debug_curline(W):-
    recorded_w(curline,cl(W,Line,Num),_),
    write(W:curline:Line:Num), nl,
    !.
debug_curline(W):-
    write('no curline for':W), nl.

debug_data(W):-
    recorded_w(W,X,_),
    write(W:'first record':X), nl,
    !.
debug_data(W):-
    write('no data for':W), nl.

%*****
%
%       Utilities
%
%*****

% recorded_w_w - a rewrite of recorded_w for a far Prolog routine.

:- mode recorded_w_w(+,?,-).

recorded_w(K,T,R):-
    key(K,Kref),
    nref(Kref,Nref),
    rec_w(Kref,Nref,R,T).

:- mode rec_w(+,+,-,?).

rec_w(K,K,_,_):-
    !,
    fail.
rec_w(K,R,R,T):-
    instance(R,T).
rec_w(K,R,Ro,T):-
    nref(R,Rn),
    rec_w(K,Rn,Ro,T).

append([], X, X).
append([H|T], L, [H|Newt]):-
    append(T, L, Newt).

member(X, [X | _]):- !.
member(X, [_ | _]):-
    member(X, _).

split(Item, List, Front, Back):-
    append(Front, [Item|Back], List),
    !.

```

```

remove(Item,List,NewList):-
    split(Item,List,Front,Back),
    append(Front,Back,NewList),
    !.

reverse(Forwards, Backwards):-
    rev(Forwards, [], Backwards),
    !.

rev([], B, B):- !.
rev([H|T], X, B):-
    rev(T, [H|X], B).

attr(black,0).
attr(blue,1).
attr(green,2).
attr(cyan,3).
attr(red,4).
attr(magenta,5).
attr(yellow,6).
attr(white,7).
attr(S:Fg:Bg, A):-
    attr(S:Fg, Af),
    attr(reverse:Bg, Ab),
    A is Af \\/ Ab,
    !.
attr(bright:X,N):-
    attr(X,A),
    N is A \\/ 8,! .
attr(reverse:X,N):-
    attr(X,A),
    rev_attr(A,N),
    !.
attr(Fg:Bg, A):-
    attr(Fg, Af),
    attr(reverse:Bg, Ab),
    A is Af \\/ Ab,
    !.

rev_attr(At, Rat):-
    BG is (At /\ 7) << 4,
    FG is (At /\ 112) >> 4,
    A is At /\ 136,
    Rat is A \\/ BG \\/ FG.

% strip leading & trailing take the leading and trailing blanks
% off a string. The final test is due to the unforgiveable nature
% of list_text to return either strings or atoms at its will
%

strip_leading(Si,So):-
    string_length(Si,Li),
    ctr_set(0,0),
    repeat,
        ctr_inc(0,Pos),
        (Pos >= Li,
         atom_string('',So)
        ;
         nth_char(Pos,Si,Char),
         Char \== 32,
         Lo is Li - Pos,

```

```

        substring(Si, Pos, Lo, So)
    ),
    !.

strip_trailing(Si, So):-
    string_length(Si, Li),
    Last is Li - 1,
    ctr_set(0, Last),
    repeat,
        ctr_dec(0, Pos),
        (Pos < 0, atom_string(' ', So)
        ;
        nth_char(Pos, Si, Char),
        Char \== 32,
        Lo is Pos + 1,
        substring(Si, 0, Lo, So)
    ),
    !.

% flag setting predicates

get_flag(F, Val):-
    recorded_w(flag, F:Val, _).

set_flag(F, Val):-
    recorded_w(flag, F:_, Ref),
    replace(Ref, F:Val),
    !.
set_flag(F, Val):-
    recorda(flag, F:Val, _).

% wraps circularly around list

next_item(Item, List, NextItem):-
    split(Item, List, Front, Back),
    (first_item(Back,
    NextItem)
    ;
    first_item(Front, NextItem)
    ),
    !.

prev_item(Item, List, PrevItem):-
    split(Item, List, Front, Back),
    (last_item(Front, PrevItem)
    ;
    last_item(Back, PrevItem)
    ),
    !.

first_item([First|_], First):-
    true.

last_item([Last], Last):- !.
last_item([H|T], Last):-
    last_item(T, Last).

error(List):-
    tget(R, C),
    tscroll(0, (24, 0), (24, 79)),

```

```
tmove(24,0),
write('*** Window Error *** '),
err(List),
tmove(R,C),
!.

err([]).
err([H|T]):-
    write(H),
    tab(1),
    err(T).
```

G Rubik

Cube Solver (rubik.pro)

```
% RUBIK.PRO
% CUBE SOLVER II
%   A Rubik's Cube Solver
%   written by Dennis Merritt
%   as described in Building Expert Systems in Prolog (Springer-Verlag)
%   available from:
%   Amzi! inc.
%   40 Samuel Prescott Dr.
%   Stow, MA 01775 USA
%   Tel 508/897-7332, FAX 508/897-2784
%   e-mail amzi@world.std.com
%
% This program may be copied, modified and redistributed although proper
% acknowledgement is appreciated.
%
% This implementation was done with Cogent Prolog, also available
% from Amzi! inc.
%
% This is the main module which contains the predicates for
%   the main control loop,
%   manual mode,
%   solve mode, and
%   utility functions.
%
% Note - The Cogent/Prolog compiler supports modules. The export declarations
% are for predicates defined in the current module which may be used
% by other modules. The import declarations are for predicates
% defined in other modules.

:-export main/0.
:-export append/3.
:-export get_flag/2.
:-export set_flag/2.
:-export error/1.
:-export reverse/2.

:-import add_history/1.      % rubhist
:-import cnd/2.             % rubdata
:-import cube_print/1.      % rubdisp
:-import get_color/1.       % rubedit
:-import pristine/1.        % rubdata
:-import rub_help/0.        % rubhelp
:-import m_disp/1.          % rubdisp
:-import m_choose/2.        % rubdisp
:-import move/3.            % rubmov
:-import orientation/2.     % rubdata
:-import pln/2.             % rubdata
:-import rdfield/2.         % rubdisp
:-import rdchar/2.          % rubdisp
:-import redit/1.           % rubedit
:-import rewrite/2.         % rubedit
:-import rot/3.             % rubmov
:-import seq/2.             % rubdata
:-import side_color/1.      % rubdata
:-import s_r/2.             % rubdata
:-import vw/2.              % rubdata
:-import wrfield/2.         % rubdisp
:-import writec/2.          % rubdisp
```

```

:-import logfile/1.      % dynamic db
:-import impplan/1.     % dynamic db
:-import state/1.       % dynamic db
:-import crit/1.        % dynamic db
:-import ghoul/1.       % dynamic db
:-import sidecolor/1.   % dynamic db
:-import flag/2.        % dynamic db
:-import cand/1.        % dynamic db
:-import candmove/1.    % dynamic db

:-op(500,xfy,:).

main :- % The start up entry point
    banner,
    go.

go:- % The main control loop
    repeat,
        init_color,
        m_disp(main), % The main menu
        m_choose(main,X), % Select an item
        do(X), % Execute it
    X == exit. % Go back to the repeat or end

% These are the predicates which are called for the various
% main menu choices. The cut after each ensures they wont be
% backtracked into when the main loop fails.

do(solve):- % in this module
    solve,
    !.
do(manual):- % in this module
    manual,
    !.
do(help):- % in rubhelp
    rub_help,
    !.
do(exit). % built-in predicate to exit

banner:-
    nl, nl,
    write($Cube Solver II$), nl,
    write($An illustrative Prolog program from$), nl,
    write($Building Expert Systems in Prolog (Springer-Verlag) by Dennis
Merritt$), nl,
    write($implemented in Cogent Prolog$), nl, nl,
    write($For more information contact:$), nl,
    write($Amzi! inc.$), nl,
    write($40 Samuel Prescott Dr.$), nl,
    write($Stow, MA 01775 USA$), nl,
    write($Tel 508/897-7332, FAX 508/897-2784$), nl,
    write($e-mail amzi@world.std.com$), nl, nl.

% These predicates initialize the state to the goal state (ghoul),
% and allow you to enter single moves. They are intended to demonstrate the
% effects of the various sequences used by the solve routines.

% They are also called by the solve routine if manual scrambling
% is requested

manual:-
    pristine(G), % Start with the goal state
    retractif(state(_)),

```

```

assert(state(G)),
cube_print(G),           % Display it
disp_moves,             % List the possible moves
repeat,                 % Start repeat-fail loop
rdfield(move,M),        % Get a move
(M == q,                % If '', clear and end
 nl,
 !,
 ;
 state(S),
 man_move(M,S,S2),      % Apply move to it
 retract(state(_)),
 assert(state(S2)),
 cube_print(S2),
 fail                    % Print it and fail back
).

man_move(M,S,S2):-
  movel(M,S,S2),
  !.
man_move(M,S,S2):-      % Pop a + in front of an unsigned move
  movel(+M,S,S2),
  !.
man_move(M,_,-):-
  error('Unknown move'-M),
  !,
  fail.

disp_moves:-            % List the three types of moves
  wrfield(moves,''),    % Heading
  move(X,_,-),          % Will backtrack through all moves
  write(X),tab(1),      % Write move
  fail.                 % Go back for the next one
disp_moves:-
  nl,
  wrfield(rotations,''), % No more moves, do the same for rots
  rot(X,_,-),
  write(X),tab(1),
  fail.
disp_moves:-            % And again for seqs
  nl,
  wrfield(sequences,''),
  seq(X,_),
  write(X),tab(1),
  fail.
disp_moves:-            % Got em all, end
  nl,
  wrfield(end_disp,'').

% This is the main body of the program, which actually solves the cube.
% See rubdoc1 and rubdoc2 for the big picture

solve:-
  m_disp(solve), % solve submenu
  m_choose(solve,X),
  rdchar(stepmode,SM),
  (SM == `y`, % check for a y (scan code 21)
   set_flag(stepmode,on)
  ;
   set_flag(stepmode,off)
  ),
  solve(X). % call solve w/ arity one with menu choice

solve(X):-

```

```

    init_solve(X), % initialize all the stuff
    T1 is cputime,
    stages,
    T is cputime - T1,
    state(S),
    cube_print(S),
    write($Done time = $),
    write(T), nl, nl.
solve(X):-
    error('failing to solve'),
    halt. % something wrong, back to main

init_solve(X):-
    wrfield(prob,X),
    initialize(X), % getting closer to the real work
    !.

initialize(X):-
    pristine(G),
    retractall(ghoul(_)),
    assert(ghoul(G)),
    init_crit(Crit), % set up the initial criteria (all variables
    retractall(crit(_)),
    assert(crit(Crit)),
    retractall(stage(_)),
    assert(stage(1)), % the first stage will call the others
    !,
    initial(X). % get specific start state in the database

initial(random):- % create a new random cube
    random_cube(Cube),
    retractall(state(_)),
    assert(state(Cube)),
    !.
initial(edit):- % edit your own
    reedit(Cube),
    retractall(state(_)),
    assert(state(Cube)),
    new_colors(Cube),
    !.
initial(manual):- % scramble your own
    manual,
    state(Cube),
    new_colors(Cube),
    !.

stages:-
    repeat,
    retract(stage(N)),
    init_stage(N,Plan), % Set the stage, get the plan
    state(S),
    cube_print(S),
    build_plan(Plan),
    improve(N,Plan), % Put the pieces in the plan in place
    vw(N,V), % undo the stage view (done by init_stage)
    undo_view(V),
    N2 is N + 1, % next stage
    assert(stage(N2)),
    N2 >= 7.

build_plan([]) :- !.
build_plan([H|T]) :-
    assert(impplan(H)),

```

```

build_plan(T).

% init_stage goes to rubdata to get the table entries which define
% the heuristics for the stage

init_stage(N,Plan):- % return list of target pieces for this stage
    wrfield(stage,N),
    cnd(N,Cands), % set up candidate moves used by search
    build_cand(Cands),
    vw(N,V), % set up preferred view for stage
    set_view(V),
    pln(N,Plan), % get list of target pieces
    !.

% improve - works through the list of target pieces for the stage.
%           it first checks to see if its already in place

improve(Stage,[]) :- !.
improve(Stage,[Piece|Rest]) :-
    impro(Stage,Piece),
    !,
    improve(Stage,Rest).

improve(Stage):-
    implan(Piece),
    impro(Stage,Piece).

impro(Stage,Piece) :-
    add_criteria(Piece,Crit), % Add new piece to criteria
    target_loc(Piece,Pos,Orient), % Where is it
    impr(Orient,Stage,Pos,Piece),
    !.

impr(0,_,_,_) :- !. % In place and oriented
impr(_,Stage,Pos,Piece) :-
    imp(Stage,Pos,Piece).

% imp - getting into the real work

imp(Stage,Pos,Piece):-
    color_piece(PieceC,Piece), % translate side notation to
    wrfield(target,PieceC), % color notation for display
    heuristics(Stage,Pos), % See if special help is needed.
    orientation(Piece,View), % Preferred view for this piece.
    set_view(View),
    crit(Crit),
    state(State),
    cntr_set(4,0), % to limit wild searches
    % gc(7),
    rotate(Moves,State,Crit), % Search for moves which transform
    retract(state(_)),
    assert(state(Crit)),
    wrfield(rot,Moves),
    add_history(Moves),
    undo_view(View),
    !.

heuristics(Stage,Pos):-
    (shift_right_1(Stage,Pos)
    ;
    shift_right_2(Stage,Pos)

```

```

    ),
    !.
heuristics(_,_) :-
    true.

% The shift_right heuristics are used to avoid the situations where
% the piece is in one of the target positions for the stage, but the
% wrong one, or mis-oriented. By blindly moving it to the right the
% search is reduced since it doesn't have to search to move it both
% out of a critical target position and back into the correct one.

shift_right_1(1,Pos):-
    smember('L',Pos), % Is the target piece already on the left?
    s_r(Pos,Moves), % If so get the canned moves to move it
    change(Moves), % right for easy search.
    !.

shift_right_2(Stage,Pos):-
    Stage < 4, % If the target piece is not on the right
    notsmember('R',Pos), % side, get the canned moves to put it
    s_r(Pos,Moves), % there to allow easier search
    change(Moves),
    !.

% rotate - the real guts of the solution, all the rest of the code provides
% support for these six lines.

% These lines illustrate the power and obscurity of Prolog.
% Prolog can be very expressive when the main information is carried
% in the predicate. However, sometimes the work is being done by
% unification, and it is not at all apparent by reading the code.
% Furthermore, since Prolog predicates often work backwards and
% forwards, it is not clear in a given case what is intended to be
% be input, and what is the output, and, as in this case, what might
% be an in-out.

% The input and output states of rotate are:

% Input: Moves - unbound
% State - bound to the cube structure for the current state
% Crit - partially bound cube structure. the bound portions
% represent the pieces in place + the current goal piece

% Output: Moves - a list of moves
% State - same as input
% Crit - fully bound to the new state

% rotate does a breadth first search by recursively calling itself
% before it calls get_move which trys new moves. it does not save the
% search trees as most breadth first algorithms do, but rather recalculates
% the moves since they can be executed so fast.

% get_move fails when called with the partially bound Crit, unless
% it is a move which reaches the desired state. The failure causes
% backtracking. However when rotate calls itself, it gives it a
% fully unbound variable NextState. This call to rotate succeeds and
% keeps adding new moves generated by get_move on backtracking.

% eventually get_move finds a match and rotate succeeds.

rotate([], State, State). % start with a no move
rotate(Moves, State, Crit):- % nothing didn't work, get serious
    rotate(PriorMoves, State, NextState), % get something to build on
    % cntr_inc(4,N4),
    % check_prog(N4),

```

```

get_move(ThisMove, NextState, Crit), % generate possible moves
append(PriorMoves, [ThisMove], Moves). % build up the list

check_prog(N) :-
    N < 250,
    !.
check_prog(_) :-
    error('not converging'),
    halt.

% The following predicates all perform various useful services
% for the main predicates above. Some are declared export as well
% and are used by other modules

% add_criteria puts a new piece on the criteria structure. it works
% by creating two piece format lists, one of the goal state, and the
% other of the current criteria. It then walks through the two lists
% simultaneously looking for the target piece in the goal state.
% when it finds it it adds it to the criteria. Crit is unbound on entry

add_criteria(Piece,Crit):-
    crit(OldCrit),
    pieces(OldCrit, OldCritP),
    ghoul(Ghoul),
    pieces(Ghoul, GhoulP),
    add_crit(OldCritP, GhoulP, NewCritP, Piece),
    pieces(Crit, NewCritP),
    retract(crit(_)),
    assert(crit(Crit)),
    !.

add_crit([V1|V2], [V3|V4], [V3|V2], V5):-
    matches(V3, V5),
    !.
add_crit([V1|V2], [V3|V4], [V1|V5], V6):-
    !,
    add_crit(V2, V4, V5, V6).
add_crit(V1, V2, V3, V4):-
    error('something wrong with add_crit'),
    !.

% The center tiles dont move on the cube. Sooo if someone enters a cube
% with different color sides then we must find the new center tiles
% and map the new colors to the sides accordingly

new_colors(Cube):-
    rewrite(ColorCube,Cube),
    get_color(ColorCube),
    rewrite(ColorCube,NewCube),
    retract(state(_)),
    assert(state(NewCube)).

% Set up the initial mapping of sides to colors

init_color:-
    side_color(SC),
    retractall(sidecolor(_)),
    ini_col(SC).

ini_col([]):- !.
ini_col([S-C|T]):-
    assert(sidecolor(S-C)),

```

```

ini_col(T).

% translate a piece in piece notation to color notation

color_piece(PieceC,Piece):-
    Piece=..[p|Args],
    col_p(ArgsC,Args),
    PieceC=..[p|ArgsC].

col_p([],[]):-!.
col_p([PC|RestC],[P|Rest]):-
    sidecolor(P-PC),
    col_p(RestC,Rest).

% execute about 50 or 60 random rotations to the goal cube. due to the
% random function, the random cubes will be the same from run to
% run. It always starts from the same seed.

random_cube(Cube):-
    ghou1(Start),
    rand_cub(Start,Cube,50).

rand_cub(Cube,Cube,0).
rand_cub(Now,Cube,N):-
    repeat,
        rand_move(M,RN),
        movel(M,Now,Next),
    NN is N - 1,
    !,
    rand_cub(Next,Cube,NN).

rand_move(M,RN):-
    RN is integer(random*12),
    arg(RN,m(+f,+b,+r,+l,+u,+d,-f,-b,-r,-l,-u,-d),M).

% the classic

member(V1, [V1|V2]):-!.
member(V1, [V2|V3]):-
    member(V1, V3).

% display a list of terms without the list notation

write_list([]):-
    true.
write_list([H|T]):-
    write(H),tab(1),
    write_list(T).

% target_loc finds the location of a given piece on the cube. it can
% also be used to find the piece at a given location. it returns the
% orientation as well, which is 0 if in place, or 1 if in place but
% twisted

target_loc(Piece, Pos, Orient):-
    ghou1(Gt),
    pieces(Gt, G),
    state(St),
    pieces(St, S),
    find_piece(G, S, Pos, Piece, Orient),

```

```

!.
target_loc(Piece, _, _):-
    error('Failing to find piece'-Piece),
    fail.

% find_piece does the work for target_loc, walking two lists simultaneously
% looking for either the piece or the position, whichever is bound.

find_piece([Gh|Gt], [Sh|St], Pos, Piece, Orient):-
    matches(Pos, Gh),
    matches(Piece, Sh),
    comp(Gh,Sh,Orient),
    !.
find_piece([V1|V2], [V3|V4], V5, V6, Orient):-
    !,
    find_piece(V2, V4, V5, V6, Orient).

matches(V1, V2):-
    comp(V1, V2, V3),
    V3 < 2,
    !.

% comp returns 0 if direct hit, 1 if in place but twisted, and
% 2 if no match

comp(p(V1), p(V1), 0):- !.
comp(p(V1, V2), p(V1, V2), 0):- !.
comp(p(V1, V2), p(V2, V1), 1):- !.
comp(p(V1, V2, V3), p(V1, V2, V3), 0):- !.
comp(p(V1, V2, V3), p(V1, V3, V2), 1):- !.
comp(p(V1, V2, V3), p(V2, V1, V3), 1):- !.
comp(p(V1, V2, V3), p(V2, V3, V1), 1):- !.
comp(p(V1, V2, V3), p(V3, V1, V2), 1):- !.
comp(p(V1, V2, V3), p(V3, V2, V1), 1):- !.
comp(V1, V2, 2).

% allows easy handling of database entries used as flags

set_flag(Flag,Val):-
    retract(flag(Flag,_)),
    assert(flag(Flag,Val)),
    !.
set_flag(Flag,Val):-
    assert(flag(Flag,Val)).

get_flag(Flag,Val):-
    flag(Flag,Val).

% get_move is used by rotate to generate moves.  the possible moves
% are stored in the database under the key cand.  backtracking causes
% successive moves to be tried

get_move(+V1, V2, V3):-
    cand(V1),
    movep(V1, V2, V3).
get_move(-V1, V2, V3):-
    cand(V1),
    movep(V1, V3, V2).

% build_cand creates the database of possible moves for a given stage.
% this is one of the important heuristics for limiting the search

```

```

build_cand(V1):-
    retractall(cand(_)),
    retractall(candmove(_)),
    build_cands(V1),
    !.

build_cands([]):- !.
build_cands([V1|V2]):-
    can_seq(V1),
    assertz(cand(V1)),
    !,
    build_cands(V2).

can_seq(M):-
    % if the search move is a sequence
    seq(M,S),
    % precompute it, so it isn't constantly
    variable(X),
    % redone during search.
    move_list(S,X,Y),
    assertz(candmove(m(M,X,Y))),
    !.
can_seq(_).

% another classic

append([], V1, V1).
append([V1|V2], V3, [V1|V4]):-
    append(V2, V3, V4).

% apply a list of moves to a state

move_list([], V1, V1):- !.
move_list([Move|V3], V4, V5):-
    movel(Move, V4, V6),
    !,
    move_list(V3, V6, V5).

% movel is the basic move predicate called from everywhere

movel(+M, V2, V3):- % distinguish between clockwise
    movep(M, V2, V3),
    !.
movel(-M, V2, V3):- % and counter clockwise moves
    movep(M, V3, V2),
    !.

% find the move, be it a simple move, a rotation, or a sequence.
% if its a sequence break it into its simple components

movep(M, X, Y):-
    move(M, X, V3),
    !,
    Y = V3.
movep(M, X, Y):-
    rot(M, X, V3),
    !,
    Y = V3.
movep(M, X, Y):-
    candmove(m(M,X,V3)),
    !,
    Y = V3.
movep(V1, V2, V3):-
    seq(V1, V4),

```

```

!,
move_list(V4, V2, V3),
!.
movep([V1|V2], V3, V4):-
    move_list([V1|V2], V3, V4),
    !.

% same as move_list, only print new state when done

move_listp(V1, V2, V3):-
    move_list(V1, V2, V3),
    wrfield(rot,V1).

% change is move_list for keeps.
% it takes the old value changes it, updates it,
% and records the history. it is called by the heuristic routines

change(ML):-
    retract(state(Old)),
    move_listp(ML,Old,New),
    add_history(ML),
    assert(state(New)),
    !.

% establish a new view. this means not just rotating the cube, but also
% rotating the criteria and the goal structures. this is necessary so
% any predicates working with any of the three winds up comparing
% apples and apples.

set_view([]):- !.
set_view(V):-
    retract(state(S1)),
    move_list(V, S1, S2),
    assert(state(S2)),
    retract(ghoul(G1)),
    move_list(V, G1, G2),
    assert(ghoul(G2)),
    retract(crit(C1)),
    move_list(V, C1, C2),
    assert(crit(C2)),
    wrfield(rot,V),
    add_history(V),
    !.

undo_view([]):- !.
undo_view(RV):-
    reverse(RV,V),
    set_view(V),
    !.

% convert a cube structure to a list of pieces and visa versa

pieces(cube(X1, X2, X3, X4, X5, X6,
            V7, V8, V9, V10, V11, V12, V13, V14, V15, V16, V17,
            V18, V19, V20, V21, V22, V23, V24, V25, V26, V27,
            V28, V29, V30, V31, V32, V33, V34, V35, V36, V37,
            V38, V39, V40, V41, V42, V43, V44, V45, V46, V47,
            V48, V49, V50, V51, V52, V53, V54
            ),
        [p(X1), p(X2), p(X3), p(X4), p(X5), p(X6),
        p(V7, V8, V9), p(V10, V11, V12), p(V13, V14, V15),
        p(V16, V17, V18), p(V19, V20, V21), p(V22, V23, V24),
        p(V25, V26, V27), p(V28, V29, V30), p(V31, V32),

```

```

        p(V33, V34), p(V35, V36), p(V37, V38), p(V39, V40),
        p(V41, V42), p(V43, V44), p(V45, V46), p(V47, V48),
        p(V49, V50), p(V51, V52), p(V53, V54)
    ]
).

% get an unbound cube

variable(cube(X1, X2, X3, X4, X5, X6,
             V7, V8, V9, V10, V11, V12, V13, V14, V15, V16, V17,
             V18, V19, V20, V21, V22, V23, V24, V25, V26, V27,
             V28, V29, V30, V31, V32, V33, V34, V35, V36, V37,
             V38, V39, V40, V41, V42, V43, V44, V45, V46, V47,
             V48, V49, V50, V51, V52, V53, V54
             )
).

% the initial criteria, unbound except for the six center tiles

init_crit(cube('F', 'R', 'U', 'B', 'L', 'D',
              V7, V8, V9, V10, V11, V12, V13, V14, V15, V16, V17,
              V18, V19, V20, V21, V22, V23, V24, V25, V26, V27,
              V28, V29, V30, V31, V32, V33, V34, V35, V36, V37,
              V38, V39, V40, V41, V42, V43, V44, V45, V46, V47,
              V48, V49, V50, V51, V52, V53, V54
              )
).

notsmember(X,Y):-
    smember(X,Y),
    !,
    fail.
notsmember(X,Y):-
    true.

% like the classic, but works on a structure instead

smember(X,Y):-
    Y=..[Fun|Args],
    member(X,Args).

% display errors

error(X):-
    wrfield(error,X), nl,
    get1(_).

% reverse a list of moves, and flip the signs along the way

reverse(L, R) :-
    rever(L, [], R).

rever([], Z, Z).
rever([H|T], X, Z) :-
    flip_sign(H, FH),
    rever(T, [FH|X], Z).

flip_sign(+ X, - X):- !.
flip_sign(- X, + X):- !.

```

```
retractif(X) :-  
    retract(X),  
    !.  
retractif(_).
```

Cube Display (rubdisp.pro)

```
% RUBDISP.PRO - Copyright (C) 1993, AmziOd
% This file contains the display predicates.

:-export cube_print/1.
:-export wrfield/2, rdfield/2, rdchar/2.
:-export writec/2.
:-export color/1,color/2,color/3.
:-export m_disp/1,m_erase/1,m_choose/2.

:-import error/1.      % rubik
:-import get_flag/2.   % rubik
:-import sidecolor/1.  % dynamic database

:- op(500,xfy,:).

% cube_print - displays the full color cube. Both variables and
%                blanks appear as spaces. unification is again used
%                to map the input cube to the individual displays
cube_print(cube(F, R, U, B, L, D,
               V7, V8, V9, V10, V11, V12, V13, V14, V15, V16, V17,
               V18, V19, V20, V21, V22, V23, V24, V25, V26, V27,
               V28, V29, V30, V31, V32, V33, V34, V35, V36, V37,
               V38, V39, V40, V41, V42, V43, V44, V45, V46, V47,
               V48, V49, V50, V51, V52, V53, V54
             )
          ) :-
    nl,
    tab(6), pc([V28, V45, V22]),
    tab(6), pc([V53, B, V51]),
    tab(6), pc([V25, V43, V19]),
    pc([V29, V54, V26, V27, V44, V21, V20, V52, V23]),
    pc([V37, L, V35, V36, U, V32, V31, R, V33]),
    pc([V17, V50, V14, V15, V40, V9, V8, V48, V11]),
    tab(6), pc([V13, V39, V7]),
    tab(6), pc([V49, F, V47]),
    tab(6), pc([V16, V41, V10]),
    tab(6), pc([V18, V42, V12]),
    tab(6), pc([V38, D, V34]),
    tab(6), pc([V30, V46, V24]),
    check_step,
    !.

check_step :-
    get_flag(stepmode, on),
    write($Hit Enter to continue$),
    get0(_).
check_step.

pc([]):-
    nl.
pc([V1| V2]):-
    sidecolor(V1 - C),
    write(C), tab(1),
    % write(V1), tab(1),
    pc(V2).

% wrfield & rdfield - allow input and output to a named field
```

```

wrfield(F,X):-
  field(F,P),
  write(P),
  write(X),
  nl.

rdfield(F,X):-
  field(F,P),
  write(P),
  read(X).

rdchar(F,X):-
  field(F,P),
  write(P),
  get(X).

% field - these are the field definitions for the cube program

field(prob, $Problem: $).
field(stage, $\nStage: $).
field(target, $Target: $).
field(rot, $Rotation: $).
field(try, $Trying: $).
field(prompt, $>$).
field(error, $Error: $).
field(done, $Done: $).
field(continue, $Hit Enter to continue.$).
field(stepmode, $Stepmode? (y/n): $).
field(history, $History? (y/n): $).
field(move, $Enter move\n(end with period, ex. u., -l., ct1., -tc3.) : $).
field(moves, $Moves: $).
field(rotations, $Rotations: $).
field(sequences, $Sequences: $).
field(end_disp, $Enter q. to end$).
field(msg20, $ $).
field(msg21, $ $).

m_disp(Menu):-
  menu(Menu, Choices),
  m_dis(1, Choices),
  !.

m_dis(_, []) :-
  nl.
m_dis(N, [H|T]) :-
  write($[$), write(N), write($)$),
  write(H), tab(1),
  NN is N + 1,
  m_dis(NN, T).

m_choose(Menu,Choice):-
  write($Choice: $),
  get(Nascii),
  N is Nascii - `0,
  menu(Menu, Choices),
  m_ch(N, Choices, Choice).

m_ch(N, [], _) :-
  write($Bad menu choice, try again$), nl,
  fail.
m_ch(1, [X|_], X) :- !.

```

```
m_ch(N, [H|T], X) :-  
  NN is N - 1,  
  m_ch(NN, T, X).  
  
menu(main, [solve, manual, help, exit]).  
menu(solve, [random, manual, edit]).
```

Cube Entry (rubedit.pro)

```
% RUBEDIT.PRO - Copyright (C) 1994, Amzi! inc.

% This module allows the user to easily enter a scrambled
% cube position.  the cube is displayed in goal form.
% the cursor keys move from tile to tile, and the f1 key
% selects the color for the tile.  repeated hits of f1
% changes the color.  f1 was chosen since that allows a
% machine with a pcmouse to do cube editing with the mouse and
% and the left button (f1) with no special changes.

:-export redit/1.
:-export set_tcolor/1.
:-export get_color/1.
:-export rewrite/2.

:-import cube_print/1.  % rubdisp
:-import error/1.      % rubik
:-import ghou/1.       % rubdata
:-import wrfield/2.    % rubdisp
:-import sidecolor/1.  % dynamic database

redit(Y):-
    ghou(G),
    cube_print(G),
    write($Enter single letters separated by spaces in the pattern$), nl,
    write($of the display.  The letters should represent the colors$), nl,
    write($on your cube.  Exact spacing isn't critical.$), nl,
    read_cube(X),      % read it off the screen
    trans_cube(X,Y),  % change colors to side notation
    cube_print(Y).
redit(_):-
    error('failing edit'),
    halt.

% read_cube - reads the edited cube directly from the screen, there was
% no need to save information about colors during the cursor movement
% stage ("edi").  it was for this reason that "change_color" writes the
% letter of the color in the tile.

% read_cube looks exactly like print_cube, only in reverse

read_cube(cube(F, R, U, B, L, D,
              V7, V8, V9, V10, V11, V12, V13, V14, V15, V16,
              V17, V18, V19, V20, V21, V22, V23, V24, V25, V26,
              V27, V28, V29, V30, V31, V32, V33, V34, V35, V36,
              V37, V38, V39, V40, V41, V42, V43, V44, V45, V46,
              V47, V48, V49, V50, V51, V52, V53, V54
              )
        ):-
    rc([V28, V45, V22]),
    rc([V53, B, V51]),
    rc([V25, V43, V19]),
    rc([V29, V54, V26, V27, V44, V21, V20, V52, V23]),
    rc([V37, L, V35, V36, U, V32, V31, R, V33]),
    rc([V17, V50, V14, V15, V40, V9, V8, V48, V11]),
    rc([V13, V39, V7]),
    rc([V49, F, V47]),
    rc([V16, V41, V10]),
    rc([V18, V42, V12]),
    rc([V38, D, V34]),
    rc([V30, V46, V24]),
    !.
```

```

rc([]):- !.
rc([V1| V2]):-
    get(X),
    name(V1, [X]),
    !,
    rc(V2).

trans_cube(X,Y):-
    get_color(X), % establish new side colors
    rewrite(X,Y). % translate color notation to side notation

get_color(X):-
    X=..[cube,F,R,U,B,L,D|_], % the sides in color notation
    set_tcolor(['F'-F, 'R'-R, 'U'-U, 'B'-B, 'L'-L, 'D'-D]).

rewrite(C,S):-
    var(S), % this one if color input and side output
    C=..[cube|Clist],
    rewr(Clist,Slist),
    S=..[cube|Slist],
    !.
rewrite(C,S):-
    var(C), % this one if side input, and color out. It
    S=..[cube|Slist], % is called by the manual routine when building
    rewr(Clist,Slist), % a cube to solve. Rotate moves might have been used
    C=..[cube|Clist], % which changed the side colors
    !.

rewrit([],[]):- !.
rewrit([X|Ctail],[Y|Stail]):-
    var(X),
    var(Y),
    !,
    rewr(Ctail,Stail).
rewrit([Color|Ctail],[Side|Stail]):-
    sidecolor(Side-Color),
    !,
    rewr(Ctail,Stail).

set_tcolor([]):- !.
set_tcolor([S-C|Tail]):-
    retract(sidecolor(S-)),
    assert(sidecolor(S-C)),
    !,
    set_tcolor(Tail).

```

Move History (rubhist.pro)

```
% RUBHIST.PRO - Copyright (C) 1994, Amzi! inc.

% This module records history information so you can unscramble
% a real cube by looking at the log file.

:-export add_history/1.

:-import append/3.      % rubik
:-import attr/2.       % rubdisp
:-import clr_bottom/0. % rubik
:-import error/1.     % rubik
:-import bug/1.       % rubik
:-import get_flag/2.  % rubik
:-import reverse/2.   % rubik
:-import move/3.      % rubmove
:-import rot/3.       % rubmove
:-import seq/2.       % rubdata
:-import wrfield/2.   % rubdisp

% add_history takes a list of moves as input. As output it sends
% the expanded version of the moves to the logfile. That is, sequences
% are broken down into primitive moves before being written to the
% window

add_history(V1):-
    expand(V1, V2),      % expand the list
    de_list(V2,V3),     % remove inbedded lists (flatten the list)
    segment_list(V3,V4), % break into pieces that fit in window
    write_hist(V4),
    !.
add_history(X):-
    error([add_history,X]).

write_hist([]).
write_hist([FirstLine|Rest]) :-
    write(' Moves: '),
    wr_hist(FirstLine),
    nl,
    write_hist(Rest).

wr_hist([]).
wr_hist([H|T]) :-
    tab(2),
    write(H),
    wr_hist(T).

% expand pushes its way through a list of moves and sequences, making
% sequences into other move lists. It takes care to preserve the
% meaning of a counterclockwise sequence by reversing the list defining
% the sequence. This reverse also changes the sign of each term along
% the way. The first argument is the input list, the second is output

expand([], []) :- !.
expand([Term|V3], [Term|V4]):-
    moveterm(Term, X), % strip the sign
    (move(X,_,_)
    ;
    rot(X,_,_) % its a primitive
    ),
    !,
```

```

    expand(V3, V4).
expand([Seq|V3], [Termlist|V5]):-
    moveterm(Seq,S), % we can guess its a sequence
    seq(S, SL),
    (signterm(Seq,-),
     reverse(SL,Sterms) % flip if necessary
    ;
     Sterms = SL
    ),
    expand(Sterms,Termlist), % double recursion, on this sequence
    !,
    expand(V3, V5). % ...and the rest of the list
expand(X,_):-
    error(['expand fails on',X]).

% separate the move and sign of a term, first arg is input, second output

moveterm(+ X, X) :- !.
moveterm(- X, X) :- !.

signterm(+ X, +) :- !.
signterm(- X, -) :- !.

% "expand" left imbedded lists where sequences used to be, flatten them
% out since they arn't necessary

de_list([], []) :- !.
de_list(V1, [V1]):-
    (V1 = +X
     ;
     V1 = -X
    ).
de_list([V1|V2], V3):-
    de_list(V1, V4), % double recursion on the head and tail
    de_list(V2, V5),
    append(V4, V5, V3).

% having flattened it, segment_list breaks a long list into smaller
% lists that will fit in the display window. This is because the
% window routine is too lazy to deal with lines that are too long

segment_list([A,B,C,D,E|Tin],[[A,B,C,D,E]|Tout]):-
    segment_list(Tin,Tout).
segment_list([],[]) :- !.
segment_list(L,[L]) :- !.

```

Moves and Rotations (rubmov.pro)

```
% RUBMOV.PRO - copyright (C) 1994, Amzi! inc.

% this file contains the definitions of all of the
% moves and rotations primitive to Rubik's Cube.

% Both moves and rotations are done using Prologs unification.
% The first argument is the name of the move or rotation, and the
% second and third arguments define transformations of the structure
% which represents the cube.

% By convention the moves are named by a single character which stands
% for the position of the side being turned. Rotations are used to
% reposition the entire cube (leaving the pieces in the same relative
% positions). They are named by the side which defines the axis
% of rotation, preceded by the letter r.

% (Why the funny variable names? This program was originally written
% in micro-Prolog (one of my favorites) with its parenthetical list
% notation. I then acquired Arity Prolog and wrote a translation
% program converted the micro-Prolog syntax to Edinburgh syntax.
% It did the dumb thing with variable names, and I've never bothered
% to fix many of them, such as these.)

% The sides are: u up, d down, l left, r right, f front, b back.

:- export move/3,rot/3.

move(u,
  cube(X1, X2, X3, X4, X5, X6, V7, V8, V9, V10, V11, V12,
        V13, V14, V15, V16, V17, V18, V19, V20, V21, V22, V23,
        V24, V25, V26, V27, V28, V29,
        V30, V31, V32, V33, V34,
        V35, V36, V37, V38, V39, V40, V41, V42, V43, V44, V45,
        V46, V47, V48, V49, V50, V51, V52, V53, V54),
  cube(X1, X2, X3, X4, X5, X6, V20, V19, V21, V10, V11,
        V12, V8, V7, V9, V16, V17, V18, V26, V25, V27, V22,
        V23, V24, V14, V13, V15, V28, V29, V30, V43, V44,
        V33, V34, V39, V40, V37, V38, V31, V32, V41, V42,
        V35, V36, V45, V46, V47, V48, V49, V50, V51, V52,
        V53, V54)
  ).
move(d,
  cube(X1, X2, X3, X4, X5, X6, V7, V8, V9,
        V10, V11, V12, V13, V14, V15, V16, V17,
        V18, V19, V20, V21, V22, V23, V24, V25, V26, V27, V28, V29,
        V30, V31, V32, V33, V34, V35, V36, V37, V38, V39, V40, V41, V42,
        V43, V44, V45, V46, V47, V48, V49, V50, V51, V52, V53, V54),
  cube(X1, X2, X3, X4, X5, X6, V7, V8, V9,
        V17, V16, V18, V13, V14, V15, V29, V28,
        V30, V19, V20, V21, V11,
        V10, V12, V25, V26, V27, V23, V22, V24, V31, V32,
        V41, V42, V35, V36, V45, V46, V39, V40, V37, V38, V43, V44, V33,
        V34, V47, V48, V49, V50, V51, V52, V53, V54)
  ).
move(r,
  cube(X1, X2, X3, X4, X5, X6, V7, V8, V9,
        V10, V11, V12, V13, V14, V15, V16, V17, V18,
        V19, V20, V21, V22, V23, V24, V25, V26, V27, V28, V29, V30, V31,
        V32, V33, V34, V35, V36, V37, V38, V39, V40, V41, V42, V43, V44,
        V45, V46, V47, V48, V49, V50, V51, V52, V53, V54),
  cube(X1, X2, X3, X4, X5, X6, V12, V11, V10, V24,
        V23, V22, V13, V14, V15, V16, V17, V18, V9,
        V8, V7, V21, V20, V19, V25, V26, V27, V28, V29, V30, V48, V47, V52,
```

```

        V51, V35, V36, V37, V38, V39, V40, V41, V42, V43, V44, V45, V46,
        V34, V33, V49, V50, V32, V31, V53, V54)
    ).
move(l,
    cube(X1, X2, X3, X4, X5, X6, V7, V8, V9,
        V10, V11, V12, V13, V14, V15, V16, V17, V18, V19, V20,
        V21, V22, V23, V24, V25, V26, V27, V28, V29,
        V30, V31, V32, V33, V34, V35, V36, V37, V38, V39, V40, V41,
        V42, V43, V44, V45, V46, V47, V48, V49, V50, V51, V52, V53, V54),
    cube(X1, X2, X3, X4, X5, X6, V7, V8, V9,
        V10, V11, V12, V27, V26, V25, V15, V14, V13, V19, V20, V21, V22, V23, V24,
        V30, V29, V28, V18, V17, V16, V31, V32, V33, V34, V54, V53,
        V50, V49, V39, V40, V41, V42, V43, V44, V45, V46, V47, V48,
        V36, V35, V51, V52, V38, V37)
    ).
move(f,
    cube(X1, X2, X3, X4, X5, X6, V7, V8, V9,
        V10, V11, V12, V13, V14, V15, V16, V17, V18, V19, V20,
        V21, V22, V23, V24, V25, V26, V27, V28, V29,
        V30, V31, V32, V33, V34, V35, V36, V37, V38, V39, V40, V41,
        V42, V43, V44, V45, V46, V47, V48, V49, V50, V51, V52, V53, V54),
    cube(X1, X2, X3, X4, X5, X6, V13, V15, V14, V7, V9, V8, V16, V18, V17,
        V10, V12, V11, V19, V20, V21, V22, V23, V24, V25, V26, V27, V28, V29,
        V30, V31, V32, V33, V34, V35, V36, V37, V38, V49, V50, V47, V48,
        V43, V44, V45, V46, V39, V40, V41, V42, V51, V52, V53, V54)
    ).
move(b,
    cube(X1, X2, X3, X4, X5, X6, V7, V8, V9,
        V10, V11, V12, V13, V14, V15, V16, V17, V18, V19, V20, V21,
        V22, V23, V24, V25, V26, V27, V28, V29,
        V30, V31, V32, V33, V34, V35, V36, V37, V38, V39, V40, V41,
        V42, V43, V44, V45, V46, V47, V48, V49, V50, V51, V52, V53, V54),
    cube(X1, X2, X3, X4, X5, X6, V7, V8, V9,
        V10, V11, V12, V13, V14, V15, V16, V17, V18, V22, V24, V23, V28,
        V30, V29, V19, V21, V20, V25, V27, V26, V31, V32, V33, V34, V35,
        V36, V37, V38, V39, V40, V41, V42, V51, V52, V53, V54, V47,
        V48, V49, V50, V45, V46, V43, V44)
    ).

rot(ru,
    cube(X1, X2, X3, X4, X5, X6, V7, V8, V9,
        V10, V11, V12, V13, V14, V15, V16, V17, V18, V19, V20, V21,
        V22, V23, V24, V25, V26, V27, V28, V29,
        V30, V31, V32, V33, V34, V35, V36, V37, V38, V39, V40, V41, V42,
        V43, V44, V45, V46, V47, V48, V49, V50, V51, V52, V53, V54),
    cube(X2, X4, X3, X5, X1, X6, V20, V19, V21, V23, V22,
        V24, V8, V7, V9, V11,
        V10, V12, V26, V25, V27, V29, V28,
        V30, V14, V13, V15, V17, V16, V18, V43, V44, V45, V46, V39,
        V40, V41, V42, V31, V32, V33, V34, V35, V36, V37, V38, V52,
        V51, V48, V47, V54, V53, V50, V49)
    ).
rot(rr,
    cube(X1, X2, X3, X4, X5, X6, V7, V8, V9,
        V10, V11, V12, V13, V14, V15, V16, V17, V18, V19, V20,
        V21, V22, V23, V24, V25, V26, V27, V28, V29,
        V30, V31, V32, V33, V34, V35, V36, V37, V38, V39, V40, V41,
        V42, V43, V44, V45, V46, V47, V48, V49, V50, V51, V52, V53, V54),
    cube(X6, X2, X1, X3, X5, X4, V12, V11,
        V10, V24, V23, V22, V18, V17, V16,
        V30, V29, V28, V9, V8, V7, V21, V20, V19, V15, V14, V13,
        V27, V26, V25, V48, V47, V52, V51, V50, V49, V54, V53, V42,
        V41, V46, V45, V40, V39, V44, V43, V34, V33, V38, V37,
        V32, V31, V36, V35)
    ).
rot(rf,
    cube(X1, X2, X3, X4, X5, X6, V7, V8, V9,

```

```
V10, V11, V12, V13, V14, V15, V16, V17, V18,  
V19, V20, V21, V22, V23, V24, V25, V26, V27, V28, V29,  
V30, V31,  
V32, V33, V34, V35, V36, V37, V38, V39, V40, V41, V42, V43, V44,  
V45, V46, V47, V48, V49, V50, V51, V52, V53, V54),  
cube(X1, X3, X5, X4, X6, X2, V13, V15, V14, V7, V9, V8, V16, V18, V17,  
V10, V12, V11,  
V25, V27, V26, V19, V21, V20, V28,  
V30, V29, V22, V24, V23, V36,  
V35, V32, V31, V38, V37, V34, V33, V49, V50, V47, V48, V53, V54,  
V51, V52, V39, V40, V41, V42, V43, V44, V45, V46)  
).
```

Rubik Help (rubhelp.pro)

```
% RUBHELP.PRO - Copyright (C) 1994, Amzi! inc.

% This is the help you get when you ask for help.

:- export rub_help/0.

rub_help:-
    helpscreen(_),
    nl, write($[more - hit any key to continue]$),
    get1(_),
    fail.
rub_help.

helpscreen(intro):-
    write($INTRODUCTION$), nl, nl,
    write($The cube solver will generate a sequence of moves that will$), nl,
    write($solve any given cube (if solvable). See rubdoc1.txt for$), nl,
    write($notes on the method.$), nl, nl.
helpscreen('menu options'):-
    write($MAIN MENU OPTIONS$), nl, nl,
    write($Solve - solves three types of cubes (from submenu)$), nl, nl,
    write($    random - generate a random cube to solve$), nl,
    write($    manual - allows you to scramble your own$), nl,
    write($    edit   - allows you to describe a real cube$), nl, nl,
    write($        with the option (prompts)$), nl, nl,
    write($    stepmode - stops after each sequence (useful if$), nl,
    write($        solving a real cube$), nl,
    write($Manual - allows manipulation of cube (useful to see the$), nl,
    write($    effects of all the legal moves$), nl, nl,
    write($Help   - this stuff$), nl, nl,
    write($Exit   - return to dos$), nl.
helpscreen(notation):-
    write($NOTES ON NOTATION$), nl, nl,
    write($The cube is unfolded so all six sides are visible. All moves$), nl,
    write($are labeled by the side they affect. The letters used are:$), nl,
    nl,
    write($          B - back$), nl,
    write($    L - left  U - up    R - right$), nl,
    write($          F - front$), nl,
    write($          D - down$), nl, nl,
    write($Directions - + clockwise, - counterclockwise$), nl, nl,
    write($Pieces are referred to by color. The colors are:$), nl, nl,
    write($    W - white, G - green, B - blue, Y - yellow,$), nl,
    write($    R - red (PC magenta), O - orange (PC red) $), nl, nl,
    write($Moves - three types$), nl, nl,
    write($    Side moves - represented by single side letter, ex +r$), nl,
    write($    Rotations - rotate entire cube, preface side with r$), nl,
    write($          ex. -ru, +rr (used to exploit symmetry)$), nl,
    write($    Sequences - sequence of moves by name ex. +ct1$), nl.
helpscreen('solve display'):-
    write($SOLVE DISPLAY FIELDS$), nl, nl,
    write($Stage - the current stage (see rubdoc1.txt)$), nl, nl,
    write($Target - the piece being solved for$), nl, nl,
    write($Trying - the n-1 nodes of the breadth first search$), nl, nl,
    write($Rotation - the chosen sequence of moves for the current goal$), nl,
    nl,
    write($Hit any key to end$).
```

Rubik Data (rubdata.pro)

```
% RUBDATA.PRO - Copyright (C) 1994, Amzi! inc.

% This file contains all the data needed to drive
% the main cube solving predicates.

:-export seq/2, s_r/2, orientation/2.
:-export cnd/2, pln/2, vw/2.
:-export pristine/1.
:-export side_color/1.

% the sequences of moves used to perform special transformations
% such as twisting the corners without moving anything else

seq(s, [+rr, -r, +l]).
seq(tc1, [-l, +u, +r, -u, +l, +u, -r, -u]).
seq(tc1u2, [+ru, +ru, +tc1, -ru, -ru]).
seq(tc3, [+r, -u, -l, +u, -r, -u, +l, +u]).
seq(ct1, [-r, +d, +r, +f, +d, -f, -u, +f,
         -d, -f, -r, -d, +r, +u]).
seq(ct3, [-r, +d, +r, +f, +d, -f, +u, +u,
         +f, -d, -f, -r, -d, +r, +u, +u]).
seq(ef1, [-u, +f, -r, +u, -f, -s, +f, -u,
         +r, -f, +u, +s]).
seq(ef2, [+l, +f, -u, +f, -r, +u, -f, -s,
         +f, -u, +r, -f, +u, +s, -f, -l]).
seq(et1, [+f, +f, +r, +r, +f, +f, +r, +r,
         +f, +f, +r, +r]).
seq(h, [+l, +f, +u, -f, -u, -l]).
seq(g, [-r, -f, -u, +f, +u, +r]).
seq(pt, [+ru, +ru]).
seq(mr2a, [+r, +f, -r, -f]).
seq(mr2b, [-r, -u, +r, +u]).
seq(mr3a, [-u, +r, +u]).
seq(mr3b, [+f, -r, -f]).

% cnd defines the moves which will be used in a given stage for search

cnd(1, [r, u, f]).
cnd(2, [r, mr2a, mr2b]).
cnd(3, [r, mr3a, mr3b]).
cnd(4, [r, tc1u2, ct1]).
cnd(5, [u, h, g, ef1, ef2]).
cnd(6, [u, tc1, tc3, ct1, ct3]).

% s_r is used by the shift_right heuristics. it lists the move sequence
% needed to move a piece which is not on the right, to the right. the
% first argument is the position the piece is at

s_r(p('F','L','U'), [-mr2a]).
s_r(p('F','L','D'), [+rr, -mr2a, -rr]).
s_r(p('B','L','U'), [-rr, -mr2a, +rr]).
s_r(p('B','L','D'), [+rr, +rr, -mr2a, -rr, -rr]).
s_r(p('F','U'), [-mr3a]).
s_r(p('F','D'), [+s, -mr3a, -s]).
s_r(p('B','U'), [-s, -mr3a, +s]).
s_r(p('B','D'), [+s, +s, -mr3a, -s, -s]).
s_r(p('L','U'), [+u, +u]).
s_r(p('F','L'), [+f, +f]).
s_r(p('L','D'), [+d, +d]).
s_r(p('B','L'), [+b, +b]).
```

```

% orientation defines the rotation moves necessary to position the
% cube to take advantage of symmetry for each piece

orientation(p('F','L','U'), []).
orientation(p('F','L','D'), [+rr]).
orientation(p('B','L','U'), [-rr]).
orientation(p('B','L','D'), [+rr, +rr]).
orientation(p('F','U'), []).
orientation(p('F','D'), [+s]).
orientation(p('B','U'), [-s]).
orientation(p('B','D'), [+s, +s]).
orientation(p('L','U'), []).
orientation(p('F','L'), [+rr]).
orientation(p('L','D'), [+rr, +rr]).
orientation(p('B','L'), [-rr]).
orientation(_, []).

% pln lists the target pieces for each stage

pln(1, [p('L','U'),p('F','L'),p('L','D'),p('B','L')]).
pln(2, [p('B','L','D'),p('F','L','D'),p('B','L','U')]).
pln(3, [p('F','U'),p('F','D'),p('B','U'),p('B','D')]).
pln(4, [p('F','L','U')]).
pln(5, [p('R','U'),p('F','R'),p('R','D'),p('B','R')]).
pln(6, [p('F','R','U'),p('B','R','U'),p('B','R','D'),p('F','R','D')]).

% vw defines the preferred orientation for a stage

vw(5, [-rf]).
vw(6, [-rf]).
vw(_, []).

% this is the pristine state

pristine(cube('F','R','U','B','L','D',
'F','R','U','F','R','D','F','L','U','F','L','D','B','R','U','B','R','D',
'B','L','U','B','L','D','R','U','R','D','L',
'U','L','D','F','U','F','D','B','U',
'B','D','F','R','F','L','B','R','B','L')).

% the initial mapping of sides and colors

side_color(['F'-'G', 'R'-'R', 'U'-'W', 'B'-'Y', 'L'-'O', 'D'-'B']).

```