**An Expert System Shell in Prolog**

The following program is a very simple expert system shell in Prolog. We introduce it both to show what a very basic shell might look like, and to introduce a few new features of Prolog. It is a slightly simpler version of the one for exercise 4, and has a different example rule base. in CS3lib.

Note that in order to define then special syntax for the rules we use some operator declarations (:-op..). This allows us to have expressions like ``if small(X) and nice(X) then good(X)''. However, these are really just Prolog facts, which can be pattern matched as normal. To get the idea of this, try loading the program and trying out the following query:

?- (if a(A) and B then C) = (if a(1) and a(2) then a(3)).

Anyway, here's the mini-shell.

%% Micro Expert System.

%% Alison Cawsey, Feb 1993

%% This is a very simple backward chaining rule-based

%% expert system. Given some possible hypotheses it backward chains on

%% each one trying to find out if it is true or not. If it cant

%% prove a fact using the rules it will just ask the user if it is

%% true or not. Once it has proved one of the hypotheses it will present

%% its conclusions to the user.

% -------------------------------------------------------------------

% OPERATOR DECLARATIONS

:- op(975, fx, if).

:- op(950, xfy, then).

:- op(925, xfy, and).

% -------------------------------------------------------------------

% EXPERT SYSTEM RULES (EXPERT KNOWLEDGE BASE)

rule(if eats(X, Y) and living(Y) then carnivore(X)).

rule(if carnivore(X) and big(X) then dangerous(X)).

rule(if has\_feathers(X) then bird(X)).

rule(if bird(X) and small(X) then good\_pet(X)).

rule(if cuddly(X) then good\_pet(X)).

% -------------------------------------------------------------------

% TEXT TEMPLATES FOR ENGLISH

% qtext/2: Provides a simple template based translation into English

% questions.

qtext(cuddly(X), ['Is ', X, ' cuddly?']).

qtext(has\_feathers(X), ['Does ', X, ' have feathers?']).

qtext(small(X), ['Is ', X, ' small']).

% atext/2: Template based translation into recommendations.

atext(good\_pet(X), ['I suggest that ', X, ' would make a good pet.']).

% ------------------------------------------------------------------

% EXAMPLE TEST CASE

find\_good\_pet :-

check\_hypothesis([good\_pet(lenny), good\_pet(eddie), good\_pet(tweety)]).

% ------------------------------------------------------------------

% MAIN EXPERT SYSTEM SHELL CODE

% check\_hypothesis(+Hypotheses)

% Succeeds when one of the hypotheses is proved true, or it

% has tried them all.

% Picks a hypothesis, and uses b\_chain to find out if it is true.

% If it is true then b\_chain succeeds and check\_hypothesis writes out the

% appropriate recommendation. If false it backtracks to 'on' to find

% another hypothesis to try.

% Once it has tried all the hypotheses it will backtrack to second

% check\_hypothesis clause and write an appropriate message.

% (Uses MacProlog built in predicates message/1 and on/2)

check\_hypothesis(Hypotheses) :-

on(Hypoth, Hypotheses), % get a member of hypotheses

bchain(Hypoth), !, % b\_chain to check if true.

atext(Hypoth, Text), % get hold of appropriate text.

message(Text). % write out the recommendation

check\_hypothesis(\_) :- message(['None of the possible hypotheses seem to be true']).

% bchain(+Goal)

% Succeeds if Goal is true, given rules + facts supplied by user as

% backward chaining proceeds.

bchain(G1 and G2):- !, % G1 and G1 are true if

bchain(G1), % G1 can be proved by backward chaining

bchain(G2). % and G2 can be too.

b\_chain(Goal) :- % G's true if its a fact!

userfact(Goal).

bchain(Goal):- % Goal is true if

rule(if Preconditions then Goal), % there's a rule concluding it

bchain(Preconditions). % and its Preconditions can be

% proved by backward chaining

bchain(Goal):- % Goal is true if

user\_says\_its\_true(Goal). % user says its true.

% user\_says\_its\_true(+Goal)

% True if there is some text to use to ask the user about it,

% and when you ask the user they say yes.

% Uses MacProlog built in predicate yesno/1

user\_says\_its\_true(Goal) :-

qtext(Goal, Text),

yesno(Text),

assert(userfact(G)). % Add the fact to Prolog's database.

The main top level Prolog predicate is check\_hypothesis. This predicate uses backtracking to go through all the members of a list of hypotheses to see whether each can be proved true by backward chaining. Once it has found one it writes out an appropriate message to the user. To do this it first looks for a bit of text appropriate for a particular conclusion (e.g., good\_pet(tweety)) using the predicate atext/2, then uses Mac Prolog's built in predicate message/1 to write out a suitable message. Message takes a single argument which should be a list, and pops up a window and writes out the items in that list.

The main b\_chainpredicate has four cases, dealing with the cases where there is a conjunction of goals to prove; where the goal to prove is just a fact; where there is a rule whose conclusion matches the goal (so its preconditions are set as new goals); and the fact where the user says that something is true. The `user\_says\_its\_true' predicate uses the Mac Prolog built in predicate yesno/1 which pops up a dialogue window with some text and two buttons (yes and no) and succeeds if the user clicks on `yes'. If the user says something IS true then that fact is asserted into the Prolog database (which will then contain things like ``userfact(has\_feathers(tweety))''). Assert is a special Prolog predicate for dynamically adding things to your prolog database. (It should be used with care, and I won't ask you to use it).

You may notice various ``!''s scattered about in the code. They are ``cuts'' and used to control backtracking. I won't ask you to know or understand cuts, but if you ever do any serious Prolog programming you will probably have to use them.

Note that ``message'' and ``yesno'' are NOT standard Prolog i/o predicates. The standard ones are write/1 which just writes out any Prolog term, and read/1 which just reads in any Prolog term (ending in full stop) and binds it to a variable. So, given the query read(A), the user can type in, say, ``fred.'' and A will be bound to the atom fred. Mac Prolog provides lots of extra window/menu/button based i/o predicates, two of which are used above.

A sample dialogue with the above system is the following (with 'S:' and 'U:' indicating system and user contributions):

?- find\_good\_pet.

S: Does lenny have feathers?

U: No.

S: Is lenny cuddly?

U: no

S: Does eddie have feathers?

U: yes.

S: Is eddie small?

U: no

S: Is eddie cuddly?

U: no

S: Does tweety have feathers?

U: yes

S: Is tweety small?

U: yes

S: I suggest that tweety would make a good pet.

The above obviously isnt adequate as an expert system shell. For a start it only allows yes/no answers; doesnt allow for rules that involve *certainties*; and doesnt have an explanation facility. If we want an explanation component for the system we need to record a trace of the rules that have been used to make a conclusion. As an absolute minimum we could just record which rules had been used, then if the user asked ``How'' the system could write out something like:

I concluded that tweety would make a good pet using the following rules:

rule: has\_feathers(X) ==> bird(X)

rule: bird(X) & small(X) ==> good\_pet(X)

So far we have given an idea of how a simple backward chaining rule based system (shell) might be written. In general an expert system shell will also allow forward chaining, and will allow some of the knowledge to be represented using a frame system. For example, we might want to have a set of frames which say (among other things) that Robbie is a white rabbit and rabbits are always cuddly. We could then check using backward chaining to find out if Robbie would make a good pet, but check using our frame system to find out if Robbie is cuddly (using inheritance from the rabbit frame).

In this section we've really looked at how we might go about writing a simple expert system *shell* - that is, the domain independent reasoning and inference bit of an expert system. However, more often than not, when you want to write an expert system you will get a commercial shell and just write the rules. Exercise 4 just involves adding domain specific rules (and facts) given an existing shell based on the one above.