KIT108 Artificial Intelligence – Prolog Workshop

Expert Systems

Sea Shell

This tutorial has been developed with reference to the following sources:
"Prolog Programming for Artificial Intelligence" (3rd Edition) by I. Bratko, Addison-Wesley, 2001.

EXPERT SYSTEM SHELLS

Expert systems can be created using AI languages such as Prolog or LISP, or by using an *expert system shell* to undertake the development. Expert systems comprise three main components. These are the:

- *knowledge-base*, which contains all of the knowledge being used by the system;
- *inference engine*, which performs the automated reasoning processes needed to derive new facts from the initial facts provided to the system; and
- *user interface*, which obtains information from the user and presents the final conclusions to the user. The user interface is also used by the knowledge engineer and the domain expert when they are developing, testing and maintaining the expert system.

After the first few expert systems were developed it was realised that the whole expert system development process could be substantially shortened if, instead of developing all new expert systems from scratch, an existing system for a similar knowledge domain had the knowledge within it removed and new knowledge inserted, but without altering the inference engine and user interface. This led to the production and sale of a number of *expert system shells*, which were essentially "empty" expert systems, consisting of just an inference engine and a user interface. The availability of these shells greatly accelerated the production of new expert systems.

CONSTRUCTING A SHELL IN PROLOG

Since we are now quite familiar with Prolog, let us construct a simple expert system shell using Prolog. Our shell will only include the inference engine component, which is quite easy to develop in Prolog, and will not have any significant user interface, which is much more difficult to construct.

To assist us in constructing the expert system shell, we also need a small knowledge-base for testing purposes. Since we are already dealing with "shells" it is fitting that we use the following rules for identifying marine creatures as our test knowledge-base:

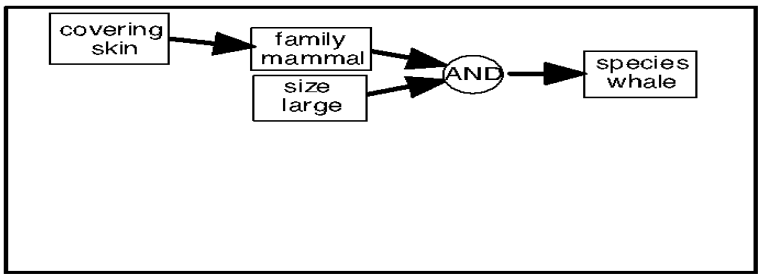
if covering_scales **then** family_fish.
if covering_skin **then** family_mammal.
if family_mammal **and** size_large **then** species_whale.
if family_mammal **and** size_small **then** species_seal.
if family_fish **and** size_large **then** species_tuna.
if family_fish **and** size_small **then** species_sardine.

Say that we are observing an animal with the following characteristics:

covering_scales, size_large

then we would want the system to tell us that we are observing a **tuna**.

1. Extend the **inference net** below to represent the Marine Creatures knowledge-base above. This diagram will be a very useful reference during the rest of the tutorial since it will help you understand how the expert system shell you are about to create operates.



Our first effort will involve developing a backward-chaining inference engine. The reason for this is that Prolog has a backward chaining facility already built into it. In fact, it would be possible to code the production rules above as Prolog rules. For example:

if family_mammal **and** size_small **then** species_seal.

could be coded as:

species(seal) :- family(mammal), size(small).

However, Prolog's own syntax for rules is not suitable for a user unfamiliar with Prolog and we want the domain expert to be able the read rules, specify new rules and modify them as easily as possible. Also we want to be able to clearly distinguish the production rules (the **knowledge-base**) from the rest of the program (the **inference engine**).

PROLOG OPERATORS

The easiest way to modify the syntax of the rules to make them easier to read is to use the Prolog operator notation. This notation allows Prolog predicates to be written in a non-standard way.

For example, say in the original **familytree** knowledge-base we had wanted to write the facts in a more readable way. Propositions such as **male(bob).** and **mother(mary, bob).** could be written as **is_male(bob).** and **is_the_mother_of(mary, bob).** but these are still convoluted sentences. It would be much better to be able to write **bob is_male.** and **mary is_the_mother_of bob.** Prolog allows predicates to be declared as operators, which enables them to be written in this way.

However, if this is done, some extra information must be supplied. Firstly, Prolog needs to know whether the operator is going to be used in:

- **prefix** notation (with the predicate appearing before the argument);
- **infix** notation (with the predicate between two arguments, such as **mary is_the_mother_of bob.**); or
- **postfix** notation (where the predicate comes after the argument, such as for **bob is_male.**)

We also need to specify operator precedence. This indicates the order in which operators are to be applied when several appear in the same expression.

new syntax

The operator definition:

:- op(800, xfx, is_the_mother_of).

will create an infix operator which will convert **mary is_the_mother_of bob** to the standard Prolog proposition **is_the_mother_of (mary, bob).** while the operator definition:

:- op(400, xf, is_male).

creates a postfix operator which will convert the proposition **bob is_male** to the standard Prolog predicate **is_male(bob).**

The initial number defines the operator's precedence. For example, say we had two operators defined as follows:

:- op(800, xfx, is_older_than).
:- op(400, fx, the_boss_of).

The **is_older_than** operator is specified with a higher precedence (800) than the **the_boss_of** operator (400). When two operators are being applied in the same statement, the operator with the lower precedence is applied within the operator with the higher precedence. This means that the statement:

the_boss_of fred is_older_than jill

is converted into the Prolog statement:

is_older_than(the_boss_of (fred), jill).

whereas, if the **is_older_than** operator were given lower precedence, it would be applied within the **the_boss_of** operator and so the same statement would be converted to:

the_boss_of(is_older_than (fred, jill)).

which is a meaningless statement in this context.

IF, THEN, AND, and OR OPERATORS

In order to create a more suitable syntax for writing production rules in Prolog, we can define **if**, **then**, **and**, and **or** as operators, as follows, making sure that you **type in** the **:-** sign at the beginning,

new syntax

```
:- op(800, fx, if).
:- op(700, xfx, then).
:- op(300, xfy, or).
:- op(200, xfy, and).
```

Notice that the **or** and **and** operators have been defined as infix operators, but with an **xfy** designation rather than an **xfx** one. This is needed to allow the system to handle a succession of these operators, all with the same precedence.

2. Construct a new Prolog file called **seashellback.pl** . First type in the operator definitions for **if**, **then**, **or** and **and** as shown above. (These must come before the production rules because they are needed to tell Prolog how to convert the rules themselves into Prolog syntax.)

3. Now type in the production rules belonging to the Marine Creatures knowledge-base. Remember to finish each rule with a full stop. **Note: You must type these rules in using the Sea Shell rule syntax (ie using if, then, and, or etc.) not the Prolog syntax**.

Prolog will take each of these rules and use the operator definitions to convert it into Prolog syntax. For example, the rule:

if family_mammal **and** size_large **then** species_whale**.**

would become:

if (**then**(**and**(family_mammal, size_large), species_whale))**.**

which is a more convoluted way of saying the same thing.

With this new higher-level syntax established, we can write the rules exactly as they are represented in everyday English.

THE FACT OPERATOR

Facts can also be represented in a slightly simpler fashion using a **fact operator** defined as follows:

:- **op(800, fx, fact).**

which will convert the the fact:

fact covering_feathers

to the Prolog proposition:

fact(covering_feathers).

4. Type in the operator definition for **fact,** as shown above, immediately under the other operator definitions in the **seashellback.pl** file.

Now that we have defined a suitable syntax for representing production rules and facts in our knowledge-base, let us proceed to construct a backward-chaining inference engine which can reason with rules and facts represented in this way.

A BACKWARD CHAINING INFERENCE ENGINE

The inference engine will be constructed using the predicate:

is_true(P).

where the proposition **P** is either given in a **fact** predicate, or can be derived using the **rules**. Since Prolog already has a backward chaining inference engine built into it, our only task is to modify the way this inference engine works so that it can work with the production rules and facts expressed using our new higher-level syntax.

Firstly we know that a proposition is true if there is a fact already in the knowledge-base specifying that proposition. For example we know that the proposition **size_large** is true if **fact(size_large)** has been asserted by the user. Since our operator notation automatically converts a fact expressed as **fact P** (where **P** is any proposition) into the Prolog syntax form **fact(P)**, then we can use the operator itself in the **is_true** predicate definition.

5. Type the Prolog clause:

is_true(P) :- fact P.

into **seashellback.pl,** immediately under the production rules. This clause says that the proposition P is true if fact P has been asserted.

We also need to incorporate the MODUS PONENS inference rule to deal with simple IF-THEN rules in the new syntax:

6. Type in the Prolog clause:

is_true(C) :- if A then C, is_true(A).

which says that if there is a rule that says "If A then C" and it is already known that antecedent A is true, then we can say that the conclusion C is true (ie. the MODUS PONENS inference rule).

Finally we need to account for the situation in which there is an **and** or an **or** between the antecedents in the rules.

7. Complete the following Prolog clause, so that it is able to determine whether the combined proposition **P1 and P2** is true, given the truth or falsity of **P1** and **P2** individually:

is_true(P1 and P2) :- .....

and type the completed clause into **seashellback.pl**.

8. Complete the following two Prolog clauses, so that they are able to determine whether the combined proposition **P1 or P2** is true, given the truth or falsity of **P1** and **P2** individually.

is_true(P1 or P2) :- .....
is_true(P1 or P2) :- .....

and type the completed clauses into **seashellback.pl**.

In order to test the inference engine, we need to insert some facts and see if appropriate conclusions are drawn. Under normal circumstances, a backward chaining inference engine would not expect these facts to be asserted before the inferencing process commences. It would begin by attempting to determine the truth of a specified goal attribute and would ask the user questions only when required. In order to do this we would need to build a user interface in Prolog and, although this can be done, it would take more time than we have available. Therefore, although our inference engine will only make use of these facts if and when required, we will need to provide all facts necessary at the beginning of the reasoning process.

9. Load the **seashellback.pl** file into Prolog using the **consult** command. Then, within Prolog, assert the two facts **covering_scales** and **size_large** as follows:

new syntax

asserta(fact covering_scales).
asserta(fact size_large).

The **asserta** command adds its parameter to the knowledgebase. It does this is a permanent way so that added facts even survives reconsulting. If you wish to "unassert" a fact you can say for example:

new syntax

retract(fact covering_scales).

If you want to retract all facts you can use either **retract(fact A).** which will show all retracted facts or **retractall(fact A).** which will silently remove them all.

10. Now type the query **is_true(species_tuna).** This will initiate the backward chaining reasoning process to determine whether species_tuna is true.

What answer do you get? Check this result against your inference net to see if this is what you expect. If not, you may need to have a closer look at your expert system shell, or your knowledge-base to see if there are any errors in it.

11. Now type **is_true(species_sardine).** Also compare this result with your inference net to see if it's what you would expect.

12. The most common way to consult an expert system is to provide the required details (observations or symptoms) and then ask the expert system to provide the result (identification or diagnosis). Do this with sea-shell by typing the query:

is_true(X).

to which Prolog will say **yes** (if it can) and give the value that has been bound to the variable **X** as its answer. You will get an infinite of answers to this question if you keep asking for more because it will give you all possible intermediate deductions as well as the final conclusion.

13. Carefully test the Marine Creatures knowledge-base by repeatedly **quit**ting Prolog (or using **retractall(fact A).**) to clear out all residual facts then starting it again and **consult**ing the **seashellback.pl** file. **Assert** appropriate facts each time and check against your inference net to ensure that all species are being identified correctly.

THE CUT FACILITY

Prolog has an automatic backtracking facility which means that if its depth-first inferencing process reaches a dead-end, it will automatically backtrack to the previous problem state and try an alternative reasoning path. This is a very useful facility but in situations where the backtracking facility is not required, it can cause inefficiency. Prolog therefore enables us to control or prevent backtracking by using the **cut** facility.

Although Prolog is not used for extensive numerical work, it is able to deal with simple numerical calculations and comparisons. A reasonably common requirement is to determine the maximum of two numbers. The following two Prolog clauses can do this:

```
maximum(X, Y, X) :- X >= Y.
maximum(X, Y, Y) :- X < Y.
```

They say that the maximum of X and Y will be X, if X is greater than or equal to Y, but will be Y if X is less than Y. Therefore the query:

```
? maximum(5, 3, Max)
```

will cause the first clause to succeed (since 5 >= 3) and Prolog's answer will be yes with Max = 5. If more answers are requested, the second clause will then be tried but will fail. On the other hand, the query:

```
? maximum(2, 7, Max)
```

will cause the first rule to fail, but the second one will succeed (since 2 < 7) and Prolog's answer will be yes, but this time with Max = 7.

These clauses work well but a closer look at them reveals some inefficiency. We know, (although Prolog doesn't) that these two clauses are mutually exclusive. If one succeeds, then the other must fail. This means that if the top rule succeeds, Prolog shouldn't backtrack to try the bottom clause since it will certainly fail. We can prevent this automatic backtracking by using the **cut** operator, represented by an exclamation mark (**!**), in the first clause.

new syntax

```
maximum(X, Y, X) :- X >= Y, ! .
```

Now, if the **?- maximum(5, 3, Max)** query is tried again, the first clause will succeed until the **cut** symbol is reached. This will automatically succeed but then will cut the backtracking process so no more alternative reasoning paths relating to the **maximum** predicate will be tried. This means that the second clause won't even be investigated, saving processing time. It also means that the second clause can be simplified, because it will only ever be reached if the first clause (X >= Y) fails, making the additional test (X < Y) unnecessary because this must always succeed if the first test fails. The two clauses now become:

```
maximum(X, Y, X) :- X >= Y, ! .
maximum(X, Y, Y) .
```

Although the cut facility can improve the efficiency of Prolog programs, it must be used with great care. It is not derived from predicate logic and so (like the **write**, **nl** and **assert** predicates) is not pure Prolog. This means that it can change the logical behaviour of Prolog programs. For example, reversing the order of the clauses in the version of **maximum** without the **cut** would have no effect on the logic of the program. Doing the same with the version with the **cut** would cause it to give an incorrect result in many instances, because maximum(X, Y, Y) would always succeed.

A FORWARD CHAINING INFERENCE ENGINE

In backward chaining we start with an hypothesis and work backwards, through the rules in the knowledge-base, to the initial facts. In many cases it can be more natural to reason in the opposite direction, from the **if** part of the rule to the **then** part. In this case the reasoning starts with the initial facts and derives any other facts that it can using the rules, until it eventually derives the conclusion required.

Constructing a simple forward chaining inference engine in Prolog is a little more complicated than constructing a backward chaining one, but not much. Given the same rule syntax as before, the rules are in the form:

If A then C.

where the antecedent **A** can contain **and**'s and **or**'s. The inference engine must take what is already known (stated in **fact** clauses) and derive all conclusions which can be drawn from these, adding these to the knowledge-base using the **asserta** command.

14. Duplicate the **seashellback.pl** Prolog file and rename it **foreshore.pl**.

15. Remove the backward-chaining inference engine (ie. the **is_true** clauses) from the file. Leave the operator definitions and the knowledge-base.

16. Begin constructing the forward chaining inference engine by defining the following clause which drives the overall inferencing process.

```
forward :-  new_derived_fact(P), !,
            write('Derived: '), write(P), nl,
            asserta(fact P),
            forward.
```

This clause will call a new_derived_fact clause which will derive a new fact from the currently available facts and rules. If this succeeds the new fact will be written out and asserted into the knowledge-base and the forward clause will be called again recursively. This process will be repeated until no more facts can be derived. The cut facility here is designed to prevent any backtracking within a single forward call, once a new fact has been derived.

17. Construct another clause which will print out an explanatory message when the forward chaining process has stopped and place this below the previous forward clause.

```
forward :-  write('No more facts.').
```

Now that the overall control of the inferencing process has been established, we need to define the **new_derived_fact** clause which applies the MODUS PONENS rule to carry out the inferencing operations. This clause will derive a fact called **C** if it finds a knowledge-base rule of the form:

if A then C.

and it finds a fact already in the knowledge-base which matches the antecedent **A**. It will only do this however, after it has determined that **C** is not already a derived fact in the knowledge-base. If it was, there would be no point in deriving it again.

18. Insert the following clause for deriving new facts using the MODUS PONENS inference rule.

new syntax

```
new_derived_fact(C) :-    if A then C,
                          \+(fact C),     /* In GProlog \+ means not */
                          composed_fact(A).
```

19. All that is left is to relate the **composed_fact** mentioned here to the facts that have been asserted into the knowledge-base. Write your own Prolog clause which states that any proposition **P** is a composed fact if that proposition is a **fact**. Refer to your backward chaining inference engine in **seashellback.pl** as a guide.

20. Also write a Prolog clause that states that a compound proposition **P1 and P2** is a composed fact if **P1** is a composed fact and **P2** is a composed fact. Refer to your backward chaining inference engine in **seashellback.pl** as a guide.

21. Then write two more clauses which indicate that **P1 or P2** is a composed fact if either **P1** is a composed fact or **P2** is a composed fact. Refer to your backward chaining inference engine in **seashellback.pl** as a guide.

Now we can test our new forward-chaining inference engine on the same Marine Creatures knowledge-base. The slight advantage we have here is that the facts are printed out as they are derived so we can clearly see the forward chaining inference process in action.

22. Load the **foreshore.pl** file into Prolog using the **consult** command. Then, within Prolog, assert the two facts **covering_scales** and **size_large**.

23. Now type **forward.** to commence the forward chaining process. What derived facts are asserted? Check the inference net to see whether these are what you would expect. Notice the order in which these facts are derived.

24. Now **quit** and **reenter** Prolog to clear out all facts. Use **consult** to load the **foreshore.pl** file again. Then assert two different facts and type **forward.** to see what marine creature is identified this time. Again check the intermediate deductions and final conclusion by referring to your inference net.

GENERATING EXPLANATIONS

An essential characteristic of an expert system is its ability to explain how it has reached a conclusion (the 'how' explanation) or why it is asking the user a particular question (the 'why' explanation). The simplest type of 'how' explanation is a list of the inferences which led from the initial facts to the conclusion. Say the **seashell.pl** expert system has concluded that the marine creature to be identified is a whale. Once this conclusion has been made, the user can ask the system how it arrived at the conclusion. The system would then say :-

> The marine creature is a whale because it is a mammal and it was large in size.
> The marine creature is a mammal because it is covered with skin.

Such as explanation is called a **proof tree**. It shows how the conclusion logically follows from the rules in the knowledge-base and the original facts provided. Let us now modify the backward chaining inference engine in **seashellback.pl** so that it is able to produce a proof tree whenever it comes up with a conclusion. The first thing to do is to define the symbol '<=' as a new infix operator.

25. Duplicate the **seashellback.pl** Prolog file and rename it **seashellex.pl**.

26. Edit **seashellex.pl**. by defining the symbol **<=** as an infix operator. Set the precedence level of this operator to **800** and define it as an **xfx** operator.

If a proposition **P** has been asserted as a fact, then the proof tree of the proposition is just **P**, the proposition itself.

27. Implement this in **seashellex.pl** by adding a proof tree component as a second argument to the corresponding **is_true** clause, so it looks like this:

is_true(P, P) :- fact P.

The first argument in the **is_true** predicate is the proposition itself while the new second argument is the proof tree of that proposition, which in this case is also just the proposition.

However, if a proposition C was derived as the conclusion of a rule then the proof tree leading to that fact is equal to the proof tree of the antecedent A of the rule, with C added.

28. Say a proposition C has been derived as the conclusion to the IF-THEN rule:

if A then C

then the proof tree of the proposition C is exactly the same as that for the antecedent A, but with a reverse arrow in front of it and the proposition C itself placed in front of that, giving :

ie. Proof Tree of C is equal to "C <= ProofTreeA"

where ProofTreeA is the proof tree of the antecedent A within the rule.
Implement this by inserting the extra proof tree argument into the is_true clause corresponding to the IF-THEN rule, thus:

is_true(C, C<= ProofTreeA) :- if A then C, is_true(A, ProofTreeA).

If **P1** and **P2** are two propositions whose proof trees are **ProofTree1** and **ProofTree2**, then the proof tree of the combined proposition **P1 and P2** is a proof tree which is just **ProofTree1 and ProofTree2.**

29. Complete the following Prolog clause, so that it is able to create the proof tree for the combined proposition **P1 and P2** given the proof trees for **P1** and **P2** individually.

is_true(P1 and P2, ProofTree1 and ProofTree2) :- .....

.....

Insert the completed clause into **seashellex.pl**.

Finally, if **P1** and **P2** are two propositions whose proof trees are **ProofTree1** and **ProofTree2**, then the proof tree of the combined proposition **P1 or P2** is a proof tree which is either **ProofTree1** or **ProofTree2,** depending on which of **P1** or **P2** is true.

30. Complete the following Prolog clauses, so that they are able to create the proof tree for the combined proposition **P1 or P2** given the proof trees for **P1** and **P2** individually. Implement this in **seashellex.pl** by inserting :-

is_true(P1 or P2, ProofTree1) :- .....

is_true(P1 or P2, ProofTree2) :- .....

Insert the completed clauses into **seashellex.pl**.

Now we need to test out the new explanation facility we have created.

31. Load the **seashellex.pl** file into Prolog using the **consult** command. Then, within Prolog, assert the two facts **covering_scales** and **size_large** as follows:

asserta(fact covering_scales).
asserta(fact size_large).

32. Now type the query **is_true(species_tuna, P).** This will initiate the backward chaining reasoning process to determine whether species_tuna is true, but this time it will create a proof tree which traces the reasoning steps being carried out. Look carefully at the proof tree (**P**) that has been created and compare it with your inference net to see if they agree. Notice that the proof tree actually represents the logical path that has been taken by the inference engine through the inference net.

33. Carefully test the explanation facility by repeatedly **quit**ting Prolog (to clear out all residual facts) then calling it again and **consult**ing **seashellex.pl** file. **Assert** appropriate facts each time and compare the proof tree which has been created with your inference net. Test that correct proof trees are created for all marine creature identifications.

THE COMPLETE SHELL

Most expert system shells permit the use of a variety of knowledge representation schemes (such as production rules and frames) and inferencing techniques (such as forward and backward chaining). Now that we have constructed and tested the forward and backward chaining inference engines and the explanation facility separately, let's combine them into one multi-purpose expert system shell.

34. Make a copy of the **seashellex.pl** file and call it **seashell.pl**. Then copy the forward chaining inference engine from **foreshore.pl** into **seashell.pl** to create a new multi-purpose forward and backward chaining shell.

35. Test out the new multi-purpose shell **seashell.pl** to ensure that it is operating correctly, using the inference net as a guide. Remember that to operate the forward chaining inference engine **assert** the required facts and then type **forward**. To operate the backward chaining inference engine with explanation facility, type in a query such as **is_true(species_seal, X).**

SEA SHELL QUESTION SHEET

NAME .....

STUDENT ID ..... LOGIN .....

Question 1.
Now we have our small Sea-Shell expert system shell, with its included Marine Creatures knowledge-base. Say you have been supplied with some extra knowledge, in the form of the following transcript from a short interview with a marine expert:

YOU	Thank you for helping me to extend my Marine Creatures expert system. Can you tell me what other types of marine creatures should have been included?
EXPERT	You should include the bird family. They differ from mammals and fish in that they are covered with feathers.
YOU	And what species of birds should we include.
EXPERT	You need to include the albatross species and the penguin species. They differ from each other in that albatrosses can fly and penguins can swim.
YOU	And what else.
EXPERT	Also you need to rearrange the mammal part of the knowledge-base. Rather than regarding whales and seals as species, we should consider them to belong to the whale and seal sub-families of the family mammal. Then include two species in each subfamily. The whale subfamily will include the killer whale and the blue whale, while the seal subfamily will include the elephant seal and the fur seal.
YOU	And how can we distinguish blue whales from killer whales and elephant seals from fur seals.
EXPERT	Blue whales feed using baleen while killer whales feed using teeth. Elephant seals make a grunting sound while fur seals make a screeching sound.
YOU	Thanks very much. I will include this valuable new knowledge into my expert system straight away.

Firstly you need to undertake some elementary knowledge analysis. Do this by drawing an extended inference net, incorporating the net drawn earlier in the tutorial, but now including the new knowledge from this transcript.

Question 2.
Use the extended inference net from Question 1 to construct extra production rules and add these new rules into the Marine Creatures knowledge-base in **seashell.pl**. Cut and paste all the production rules that you have now inserted into the **seashell.pl** system in the space below (or you could just show your tutor in the tutorial/workshop).

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Question 3.
Test out the extended Marine Creatures knowledge-base by asserting the facts:

covering_feathers. **action_swims.**

and using the forward chaining inference engine on the facts given, writing down the intermediate deductions and the final conclusion reached. Check these against the inference net in question 1.

.....

Derived: family_bird
Derived: species_penguin

.....

.....

Given the conclusion reached above, use the explanation facility to create a the proof tree justifying that conclusion. Remember to quit and reenter Prolog, before doing this, to ensure all previously deduced facts have been removed. Check the proof tree produced against the inference net in question 1.

.....

P = (species_penguin<=(family_bird<=covering_feathers) and action_swims)

.....

.....

.....

Question 4.
Given that you have seen an animal which is covered with skin, makes a screeching sound and is small in size, assert appropriate facts into **seashell.pl** and then use the forward chaining inference engine to identify the species of marine creature being observed. Write the name of the species here :

species_fur_seal

.....

Given the conclusion reached above, use the backward chaining with explanation facility to create a proof tree justifying that conclusion. Check this proof tree against the inference net in question 1.

.....

P = (species_fur_seal<=(subfamily_seal<=(family_mammal<=covering_skin) and size_small) and sound_screech)

.....

.....