CSE1101 Discrete Mathematics for Computer Science

Assignment

This assignment has been adapted from the book *Building Expert Systems in Prolog* by Dennis Merritt. While the book can be distributed for non-profit purposes, according to copyright restrictions, the distribution must be a true and complete copy of the original material. However, for the purposes of this assignment, there is need to extract Chapters 1 and 2 and modify some of the content therein. You will find the extracted and modified sections in this document, and it is to be considered unofficial companion notes to the original text.

Assignment Type: Individual

Instructions: You are required to build an expert system in Prolog. You will find details in **SECTION 3**. However, you are required to read the overview in **SECTION 1** and complete the tutorial in **SECTION 2** before attempting the assignment question.

Submission Details are included at the end of this document.

Section 1: Expert Systems Overview

Expert systems are computer applications or a collection of applications that solve problems which require expert or specialist knowledge from a particular domain. These systems are commonly used for diagnosis and classifications in areas such as medicine, financial planning, biological classification, computer configuration and troubleshooting.

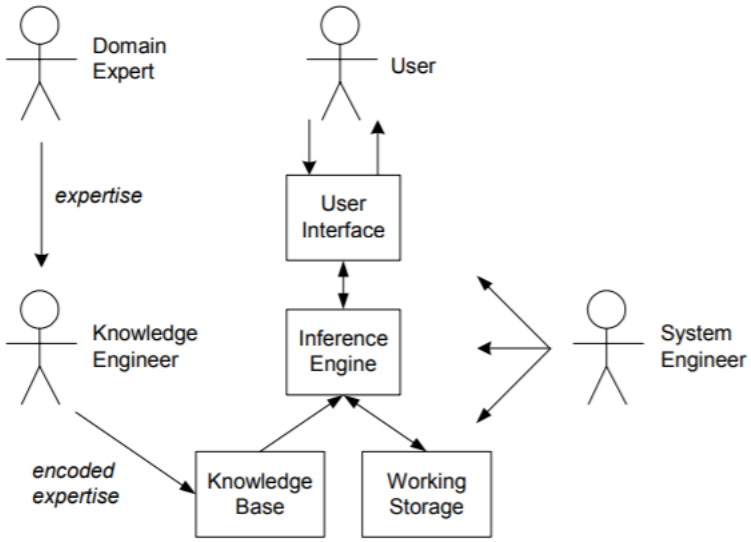

Figure 1 Expert System Components and Human Interfaces

As illustrated in Figure 1, the major components of an expert system are the:

- Knowledge base – a representation of the expertise typically expressed using IF THEN rules;
- Working storage – a type of temporary database that holds data about the program being solved;
- Inference engine – the logical core of the system which links rules and facts to deduce new information.
- User interface – an interface between the (non-expert) user and the system.

To understand how these systems are designed, it is necessary to understand the major roles of the different individuals interacting with the system. As illustrated in Figure 1, these roles include the:

- Domain expert – the individual who is the expert in solving the problems in the domain-specific area e.g. doctor, biologist;
- Knowledge engineer – the individual who encodes expert knowledge from the domain expert into the knowledge base;

- (Non-expert) User – the individual who will consult the system to get advice provided by the domain expert;
- System engineer – the individual who builds the user interface, designs the declarative format of the knowledge base, and implements the inference engine.

Section 2: Tutorial

You are required to complete this tutorial before attempting the assignment question. This is a tutorial on creating a simple expert system with a user interface. It has been adapted from Merritt's (2012) Bird Identification System which can be used to identify a small subset of birds.

2.1 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,
...
```

2.2 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 (a type of bird) 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[1] **`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).
```

Enter these rules about the birds in a Prolog file (.pl) named 'birds'.

In order for the bird rules to succeed in distinguishing between types of birds, we would have to store facts about a particular bird that can later be identified in the program. For example, if we added the following facts to the program:

```
family(albatross).

color(dark).
```

the following query could be used to identify the bird:

```
?- bird(X).

X = black_footed_albatross
```

Add the facts above to your Birds.pl file and then verify that you get the output `X = black_footed_albatross` ***when you perform the query. Ensure you save the file after adding the facts.***

Note: the user interface is the Prolog interpreter's interface, and the input data (facts about family and color) is stored directly in the program.

[1] Note **`bird/1`** means the predicate bird accepts 1 argument

2.3 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).
```

Enter the above rules about family and order below the last rule in your current Birds.pl file.

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

```
nostrils(external_tubular).

live(at_sea).

bill(hooked).

size(large).

wings(long_narrow).

color(dark).
```

In your Birds.pl file, replace the previously entered facts about family and color at the end of the file with the facts listed above.

You will find that the same query still identifies the bird:

```
?- bird(X).

X = black_footed_albatross
```

Re-consult Birds.pl and verify that you get the output `X = black_footed_albatross`

2.4 User Interface

The system can be improved by providing a user interface which prompts for information when it is needed – rather than having it entered as facts beforehand. The predicate **ask** will provide this functionality.

2.4.1 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 laysan_albatross if it has the following values for these selected bird attributes:

Attribute	**Value**
family	albatross
color	white

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 laysan_albatross had the condition **color(white)** in the rule.

2.4.2 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 which call ask.

```
color(X):- ask(color, X).
wings(X):- ask(wings, X).
size(X):- ask(size, X).
bill(X):- ask(bill, X).
live(X):- ask(live, X).
nostrils(X):- ask(nostrils, X).
voice(X):- ask(voice, X).
flight(X):- ask(flight, X).
feet(X):- ask(feet, X).
neck(X):- ask(neck, X).
```

At the end of your Birds.pl file, enter the above as askable rules.

We now need to implement the **ask** predicate. 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).
```

Enter the ask predicate at the end of the list of askable rules you just inserted and delete all the facts that are currently in the file.

The **read** in the **ask** predicate will succeed if the user answers "yes", and fail if the user types anything else. **At this stage the program can be run and give useful responses without having any facts built into it.** This is why we could have deleted all the facts we had stored.

With this version of the program, the same query **`?- bird(X)`** 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
```

Re-consult Birds.pl and verify that your program runs the same as the above snippet. Note, you would have to put a period after 'yes' and hit the ENTER key to move to the next question.

Now, 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 a process known as '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.

Try this yourself and see if you experience the same issue.

```
?- 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 ;

?- bird(X).
nostrils:external_tubular? yes.
live:at_sea? |: yes.
bill:hooked? |: yes.
size:large? |: yes.
wings:long_narrow? |: yes.
color:white? |: no.
nostrils:external_tubular? |: 
```

2.4.3 Remembering the answer

A 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 **assert**ed 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. Do not focus too much on the syntax of this code. It is sufficient to understand its purpose (HINT: this snippet of code can be re-used as is in your assignment).

Replace the previous version of the ask rule with this new version of ask.

An important step here is to declare to Prolog that **known/3** is a dynamic predicate. To do this, insert the following:

:-dynamic known/3.

at the very top of your Birds.pl file.

Re-consult the program. Now, if you answer "no" to the last question, you will not be asked questions that you have already answered - as shown below.

```
?- bird(X).
nostrils:external_tubular? : yes.
live:at_sea? : |: yes.
bill:hooked? : |: yes.
size:large? : |: yes.
wings:long_narrow? : |: yes.
color:white? : |: no.
color:dark? : |:
```

However, you will notice that after Prolog identifies a bird, you will not be able to use the system to identify any other birds. This is because the previous bird is bound to the known variable. Observe the snippet below:

```
?- bird(X).
nostrils:external_tubular? : yes.
live:at_sea? : |: yes.
bill:hooked? : |: yes.
size:large? : |: yes.
wings:long_narrow? : |: yes.
color:white? : |: no.
color:dark? : |: yes.

X = black_footed_albatross ,

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

Enter the series of inputs above and observe if you are in the same situation.

To correct this, we need to perform some housekeeping using the following code:

```
solve :-
retractall(known(_,_,_)),
top_goal(X),
write('The bird is '), write(X), nl.

solve :-
write('This is an unknown bird.'), nl.
```

Enter this code at the end of your Birds.pl file. Like the previous ask code, this can be used as is in your assignment – with of course modifications to the parts related to 'bird'.

In summary, this bit of code will start a consultation when the user enters **`solve.`** and remove any previous **`known`**s when a new consultation is triggered (**`retractall`**).

Note that the code refers to a **`top_goal(X).`** This will trigger a query which we will specify as **`?-bird(X)`**. We therefore do not need to type ?-**`bird(X)`** to run the system. We can instead type **`solve.`**

To specify the top goal, enter the following line at the top of the Birds.pl file `below` **`:- dynamic known/3).`**

top_goal(X) :- bird(X).

If you have done everything correctly, you will be able to call solve every time you want to query a new bird.

Section 3: Instructions

In this assignment, you will build an expert system in Prolog using the knowledge base provided.

Knowledge Base:

Mr. John is an expert in the field of biology, and has many years of experience studying and classifying plants and animals. The following documents Mr. John's expert knowledge in the field:

> In the animal kingdom, all animals are classified as vertebrates or invertebrates. Invertebrates are animals with an absent vertebral_column, while vertebrates are defined by a present vertebral_column.
>
> Apart from this, vertebrates have an internal skeleton, while invertebrates have an external skeleton. Also, the nerve_cord in a vertebrate is dorsal_and_hollow while it is ventral_and_solid for invertebrates. In terms of heart_location, for vertebrates, their heart is located to the right_side of the body, while it is to the dorsal_side for invertebrates. Their blood also differs. For the vertebrates, haemoglobin is present in red_blood_cells, while it is dissolved_in_plasma for invertebrates.
>
> Of course, the animals will also have observable features such as their unique sound and color. While we can broadly identify an animal as vertebrate and invertebrate, these features can tell us exactly what animal it is. For example, while cows are vertebrates, they are also black_and_white, and they make the sound moo. Similarly, invertebrate mosquitoes are black and they make the sound zzz.
>
> The following tables more comprehensively capture Mr. John's knowledge:

Type	**Vertebral Column**	**Skeleton**	**Nerve Cord**	**Heart Location**	**Haemoglobin**
Vertebrate	Present	Internal	Dorsal and hollow	Right	Red blood cells
Invertebrate	Absent	External	Ventral and solid	Dorsal	Dissolved in plasma

Animal	**Type**	**Color**	**Sound**
Cow	Vertebrate	Black_and_white	Moo
Mosquito	Invertebrate	Black	Zzz
Lion	Vertebrate	Brown	Rawr
Bee	Invertebrate	Yellow	Buzz
Crickets	Invertebrate	Green	Chirp

Your Tasks

1. **Create the Knowledge Base**
    a. Convert the information given in the Knowledge Base above into Prolog facts and rules.
2. **Build the User Interface**
    a. Use predicates from Section 2 to create a similar simple question–answer interface, to determine the type of animal.
3. **Add Clear Comments**
    a. Write a comment for every important line or block of code.
    b. Explain what the code does in your own words.
    c. Use the standard Prolog comment format:

```
% This rule determines whether an animal is a vertebrate.
```

4. **Unique Style / Program Extension / Interpretation**
    a. You must add your own creative element to the program. This could be, for example improving the user interface (e.g., more guided prompts), or formatting results in a clearer or more user-friendly style.

Mark Scheme

Working Program / Program Structure	40%
Detailed / Accurate Documentation	50%
Unique Style/ Program Extension / Interpretation	10%

Submission Details

Available From: 26th November, 2025

Due: 10th December, 2025 before 23:59 hrs

If a student submits an assignment late without being granted an extension, 20% of the total points will be deducted for each day the assignment is late.

Submission Medium: UG Moodle ONLY. Assignments submitted through alternative channels will be disregarded.

File Name and Type: Assignments that do not follow these standards will be penalized.

CSE1101_Assignment_FirstNameStudent_LastNameStudent.pl

You are only required to submit the .pl file. All comments must be in the .pl file. No other file types will be accepted.

Plagiarism and use of Generative AI:

The use of generative AI is prohibited for this assignment.

Additionally, please note that the University's plagiarism rules and policies also apply to coding assignments:

- If there is evidence to suggest collusion with classmates, the assignment will be flagged as plagiarized.
- If you insert any body of code into the assignment that you did not write, the assignment will be flagged as plagiarized.
- If you submit an assignment that is not your own/you did not do, the assignment will be flagged as plagiarized.

Please note all plagiarized assignments or assignments containing AI generated material will be awarded a score of zero (0). Instances of plagiarism will also be formally reported to your department and faculty which can lead to further disciplinary actions.

Reference

Merritt, D. (2012). *Building expert systems in Prolog*. Springer Science & Business Media.

Questions

If you have any questions about this assignment, contact alicia.layne@uog.edu.gy