Module 7

Knowledge Representation and Logic – (Rule based Systems)

Lesson 18

Rule based Systems - II

Version 2 CSE IIT, Kharagpur

7.2.5 Programs in PROLOG

These minimal notes on Prolog show only some of its flavor.

Here are facts

```
plays(ann,fido).
friend(john,sam).
```

where ann, fido, john, and sam are individuals, and plays and friend are **functors**. And here is a rule

```
likes(X,Y) :- plays(X,Y),friend(X,Y).
```

It says that if X plays with Y and X is a friend of Y then X likes Y. Variables start with capital letters (If we are not interested in the value of a variable, we can just use _ (underscore)).

In a rule the left-hand side and the right-hand side are called respectively the **head** and the **tail** of the rule.

The prompt in prolog is

```
| ?-
```

You exit prolog with the statement

```
halt.
```

You can add rules and facts to the current session in a number of ways:

1. You can enter clauses from the terminal as follows:

```
| ?- consult(user).
| like(b,a).
| like(d,c).
^D
```

   which adds the two clauses to the working space.

6. Or you can read in a file:

```
| ?- consult('filename').
```

   which is added at the end of the working space.

8. Or you can assert a clause

```
| ?- assert(< clause >).
```

Here are some confusing "equalities" in prolog:

```
                          variable        arithmetic
predicate  relation       substitution    computation
-----------------------------------------------------------
==         identical      no              no
=          unifiable       yes            no
=:=        same value     no              yes
is         is the         yes             yes
           value of
```

and some examples of their use:

```
?- X == Y.
no

?- X + 1 == 2 + Y.
no

?- X + 1 = 2 + Y.
X = 2
Y = 1

?- X + 1 = 1 + 2.
no
```

Example: Factorial1

```
factorial(0,1).
factorial(N,F) :- N>0, N1 is N-1, factorial(N1,F1),F is N*F1.
```

then

```
?- factorial(5,F).
F = 120

?- factorial(X,120).
instantiation error
```

Example: Factorial2: dropping N>0 from factorial1

```
factorial(0,1).
factorial(N,F) :- N1 is N-1, factorial(N1,F1),F is N*F1.
```

then

```
?- factorial(5,F).
F = 120;                    Here ";" asks for next value of F
keeps on going until stack overflow
```

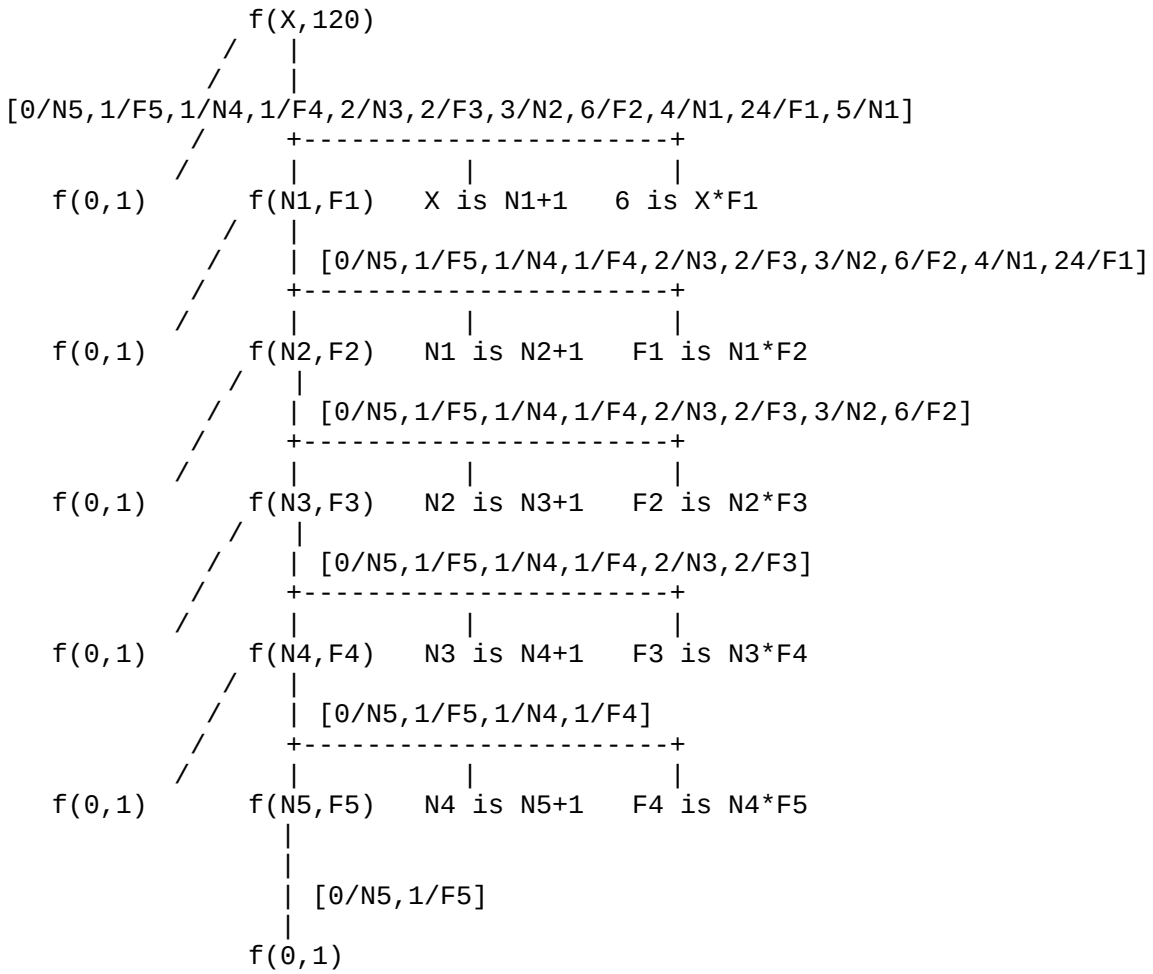
Example: Factorial3: Changing order of terms

```
(1)     factorial(0,1).
(2)     factorial(N,F) :- factorial(N1,F1), N is N1+1, F is N*F1.
```

```
then
        ?- factorial(5,F).
        F = 120

        ?- factorial(X,120).
        X = 5;
        integer overflow

Here is why factorial(X,120) returns 5. For brevity, instead of
"factorial"
we will write "f".
```



In this diagram we see the substitutions computed. Much is not said in the diagram, for example why we abandon the unifications with the various f(0,1)s. [Let's say it for the second f(0,1) from the top: because it forces the substitution [0/N1,1/F1,1/X] and this cause 6 is X*F1 to fail.]

Lists

Lists are very much as in lisp. In place of Lisp's cons, in Prolog we use the "." or dot:

```
Dot Notation           List Notation   Lisp Notation
-----------------------------------------------------
.(X,Y)                 [X | Y]         (X . Y)
.(X, .(Y,Z))           [X,Y|Z]         (X (Y . Z))
.(X, .(Y, .(Z, [])))   [X,Y,Z]         (X Y Z)
```

Example: len

```
len([],0).
len([_|T], N) :- len(T,M), N is M+1.

?- len([a,b,c],X).
X = 3

?- len([a,b,c], 3).
yes

?- len([a,b,c], 5).
no
```

Example: member

member(X,Y) is inteded to mean X is one of the top level elements of the list Y.

```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).

?- member(X, [1,2,3,4,5]).
X=1;
X=2;
X=3;
X=4;
X=5;
no
```

Example: select

select(X,A,R) is intended to mean that X is a top level element of the list A and that R is what is left of A after taking X out of it.

```
select(H,[H|T],T).
select(X,[H|T],[H|T1]) :- select(X,T,T1).

?- select(X,[a,b,c],R).
X=a
R=[b,c];
X=b
R=[a,c];
X=c
R=[a,b];
No
```

The Cryptography Problem

Here is a problem:

```
  S E N D +
  M O R E
---------
M O N E Y
```

to be solved by mapping the letters into distinct digits and then doing regular arithmetic. We add variables to represent the various carries:

```
  C3 C2 C1
  S  E  N  D +
  M  O  R  E
------------
M O  N  E  Y
```

We observe that carries can only be 0 or 1 and thus that M has to be 1. Then here is a solution:

```
solve([S,E,N,D], [M,O,R,E], [M,O,N,E,Y]) :-
    M=1, L=[2,3,4,5,6,7,8,9],
    select(S,L,L1), S>0, (C3=0; C3=1),        ";" means OR
    O is S+M+C3-10*M, select(O, L1, L2),
    select(E,L2,L3), (C2=0;C2=1),
    N is E+O+C2-10*C3, select(N,L3,L4), (C1=0;C1=1),
    R is E+10*C2-(N+C1), select(R,L4,L5),
    select(D,L5,L6),
    Y is D+E-10*C1, select(Y,L6,_).

?- solve([S,E,N,D], [M,O,R,E], [M,O,N,E,Y]).
    S=9
    E=5
    N=6
    D=7
    M=1
    O=0
    R=8
    Y=2;
    No
```

7.2.6 Expert Systems

An **expert system** is a computer program that contains some of the subject-specific knowledge of one or more human experts. An expert systems are meant to solve real problems which normally would require a specialized human expert (such as a doctor or a minerologist). Building an expert system therefore first involves extracting the relevant knowledge from the human expert. Such knowledge is often heuristic in nature, based on useful ``rules of thumb" rather than absolute certainties. Extracting it from the expert in a way that can be used by a computer is generally a difficult task, requiring its own expertise. A *knowledge engineer* has the job of extracting this knowledge and building the expert system *knowledge base*.

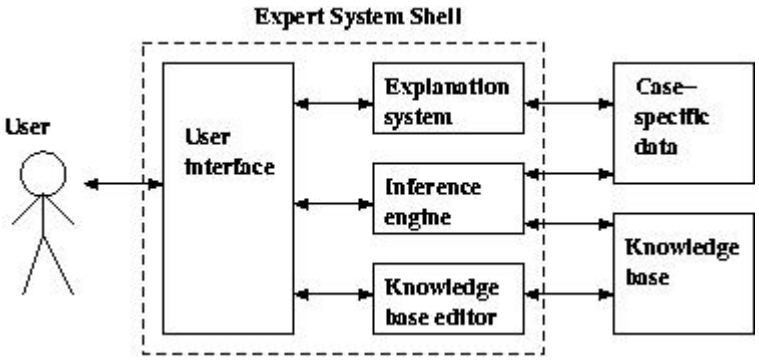
A first attempt at building an expert system is unlikely to be very successful. This is partly because the expert generally finds it very difficult to express exactly what knowledge and rules they use to solve a problem. Much of it is almost subconscious, or appears so obvious they don't even bother mentioning it. *Knowledge acquisition* for expert systems is a big area of research, with a wide variety of techniques developed. However, generally it is important to develop an initial prototype based on information extracted by interviewing the expert, then iteratively refine it based on feedback both from the expert and from potential users of the expert system.

In order to do such iterative development from a prototype it is important that the expert system is written in a way that it can easily be inspected and modified. The system should be able to explain its reasoning (to expert, user and knowledge engineer) and answer questions about the solution process. Updating the system shouldn't involve rewriting a whole lot of code - just adding or deleting localized chunks of knowledge.

The most widely used knowledge representation scheme for expert systems is rules. Typically, the rules won't have certain conclusions - there will just be some degree of certainty that the conclusion will hold if the conditions hold. Statistical techniques are used to determine these certainties. Rule-based systems, with or without certainties, are generally easily modifiable and make it easy to provide reasonably helpful traces of the system's reasoning. These traces can be used in providing explanations of what it is doing.

Expert systems have been used to solve a wide range of problems in domains such as medicine, mathematics, engineering, geology, computer science, business, law, defence and education. Within each domain, they have been used to solve problems of different types. Types of problem involve *diagnosis* (e.g., of a system fault, disease or student error); *design* (of a computer systems, hotel etc); and *interpretation* (of, for example, geological data). The appropriate problem solving technique tends to depend more on the problem type than on the domain. Whole books have been written on how to choose your knowledge representation and reasoning methods given characteristics of your problem.

The following figure shows the most important modules that make up a rule-based expert system. The user interacts with the system through a *user interface* which may use menus, natural language or any other style of interaction). Then an *inference engine* is used to reason with both the *expert knowledge* (extracted from our friendly expert) and data specific to the particular problem being solved. The expert knowledge will typically be in the form of a set of IF-THEN rules. The *case specific data* includes both data provided by the user and partial conclusions (along with certainty measures) based on this data. In a simple forward chaining rule-based system the case specific data will be the elements in *working memory*.



Almost all expert systems also have an *explanation subsystem*, which allows the program to explain its reasoning to the user. Some systems also have a *knowledge base editor* which help the expert or knowledge engineer to easily update and check the knowledge base.

One important feature of expert systems is the way they (usually) separate domain specific knowledge from more general purpose reasoning and representation techniques. The general purpose bit (in the dotted box in the figure) is referred to as an *expert system shell*. As we see in the figure, the shell will provide the inference engine (and knowledge representation scheme), a user interface, an explanation system and sometimes a knowledge base editor. Given a new kind of problem to solve (say, car design), we can usually find a shell that provides the right sort of support for that problem, so all we need to do is provide the expert knowledge. There are numerous commercial expert system shells, each one appropriate for a slightly different range of problems. (Expert systems work in industry includes both writing expert system shells and writing expert systems using shells.) Using shells to write expert systems generally greatly reduces the cost and time of development.

Questions

1. Consider the first-order logic sentences defined below.

$\forall x,y\ P(x,y) \wedge Q(y,x) \Rightarrow R(x,y)$
$\forall x,y\ S(x,Bob) \wedge S(y,x) \Rightarrow P(x,y)$
$\forall x,y\ S(x,y) \Rightarrow Q(y,x)$
$\forall x,y\ T(x,y,x) \Rightarrow Q(x,y)$
$\forall x,y\ T(x,x,y) \Rightarrow Q(x,y)$
T(Alice,Dawn,Alice)
T(Eve,Carl,Eve)
T(Alice,Bob,Dawn)
T(Carl,Carl,Alice)
S(Bob,Alice)
S(Carl,Bob)
S(Dawn,Carl)
S(Carl,Dawn)
S(Alice,Dawn)
S(Eve,Carl)

Use backward chaining to find **ALL** answers for the following queries. When matching rules, proceed from top to bottom, and evaluate subgoals from left to right.

Query 1: $\exists x\ Q(Alice, x)$

Query 2: $\exists x,y\ R(x,y).$

2. Translate the following first-order logic sentences into Prolog. Note, some sentences may require more than one Prolog statement.

a) Knows(Sylvester,Tweetie)
b) $\forall x,y$ Friend(x,y) $\Rightarrow$ Knows(x,y)
c) $\forall x,y$ (Cat(x) $\wedge$ Bird(y)) $\Rightarrow$ LikesToEat(x,y)
d) $\forall x$ (Parakeet(x) $\vee$ Penguin(x)) $\Rightarrow$ Bird(x)

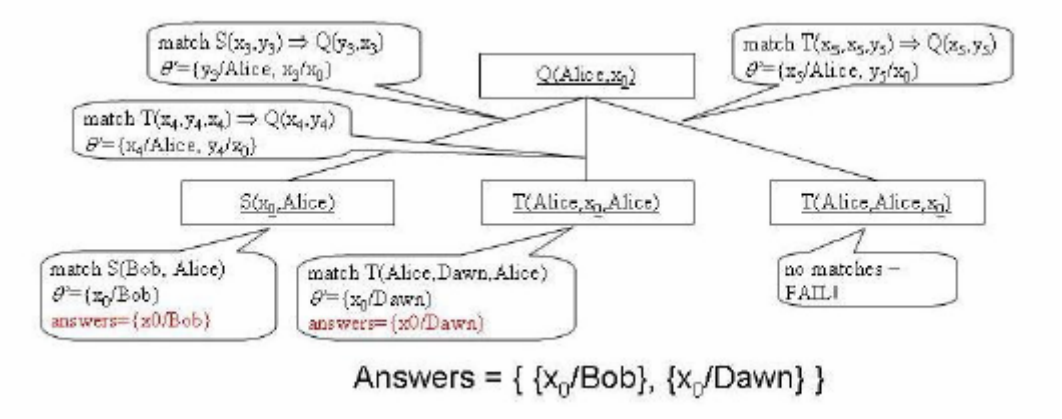
e) $\forall x$ Parakeet(x) $\Rightarrow$ (Flies(x) $\wedge$ Chirps(x))

3. Write a PROLOG programs to append two lists.

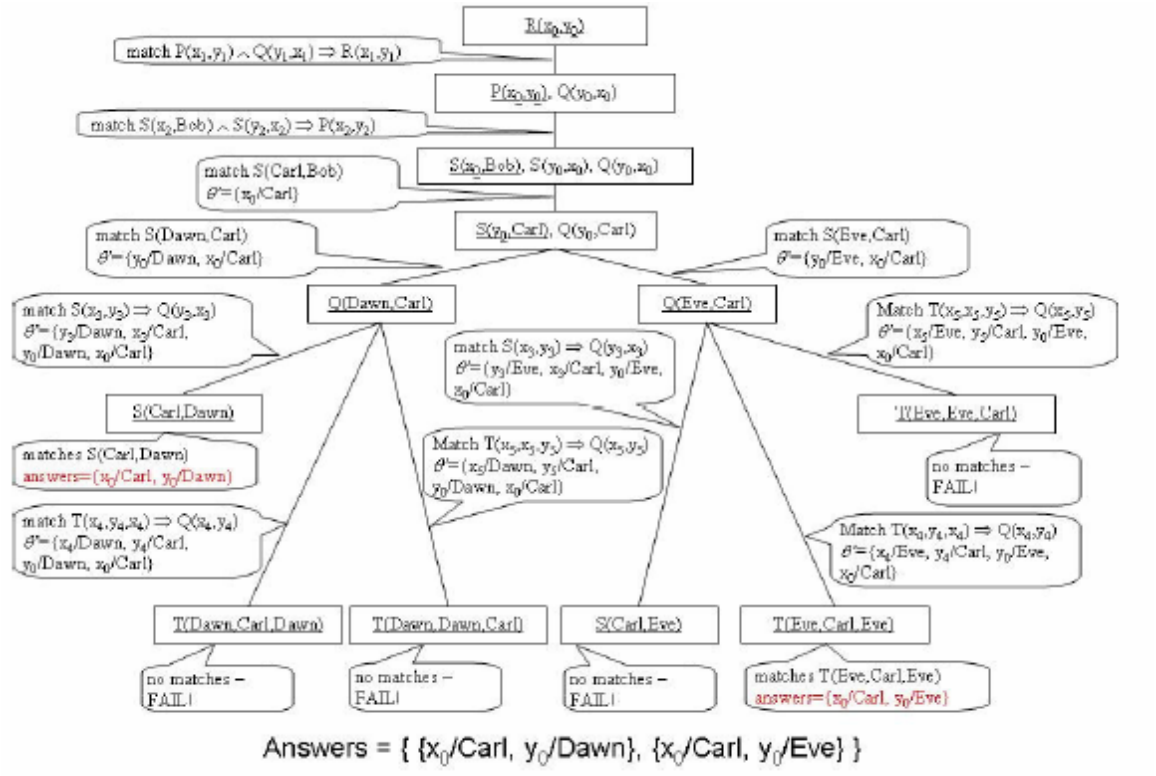
Solution

1. The proof trees are shown below:

Query 1:



Query 2:



2. Prolog statements are

a. *knows(sylvester,tweetie).*

b. *knows(X,Y):- friend(X,Y).*

c. *likesToEat(X,Y) :- cat(X), bird(Y).*

d. *bird(X) :- parakeet(X).*
bird(X) :- penguin(X).

e. *flies(X) :- parakeet(X).*
chirps(X) :- parakeet(X).

3. Prolog program

```
append(nil,L,L).
append(c(X,L),M,c(X,N)) :- append(L,M,N).
```