CS5100: Foundations of Artificial Intelligence

Encoding Logic from Text

Dr. Rutu Mulkar-Mehta
Lecture 8

Administrative

- Today
  - Last class on Logic and KR
  - In class assignment today
  - There have been a lot of questions on Prolog and Pyke, and today we will go through the basics of both
  - We will be working with SWI-Prolog as we go through the slides.
- Next week
  - Statistical AI

Proposition Logic in Prolog - 1

- In Prolog we can make some statements by using facts.
- Facts either consist of a particular item or a relation between items.
- For example we can represent the fact/proposition that it is sunny by writing the program:

```
sunny.
```

- We can now ask a query of Prolog by asking

```
?- sunny.
```

?- is the Prolog prompt. To this query, Prolog will answer yes. sunny is true because (from above) Prolog matches it in its database of facts.

Prolog – Facts' Syntax

- Facts/propositions should always begin with a lowercase letter (constant) and end with a full stop.
- The facts themselves can consist of any letter or number combination, as well as the underscore _ character.
- However, names containing the characters -, +,*,/, or other mathematical operators should be avoided.

Proposition Logic in Prolog - 2

Examples of Simple Facts

Here are some simple facts about an imaginary world. /* and */ are comment delimiters

```
john_is_cold.                 /* john is cold */
raining.                      /* it is raining */
john_Forgot_His_Raincoat.     /*  john forgot his raincoat */
fred_lost_his_car_keys.       /* fred lost is car keys */
peter_footballer.             /* peter plays football */
```

Proposition Logic in Prolog - 3

These describe a particular set of circumstances for some character john. We can interrogate this database of facts, by again posing a query. For example: *{note the responses of the Prolog interpreter are shown in italics}*

?- john_Forgot_His_Raincoat.
yes

?- raining.
yes

?- foggy.
no

Proposition Logic in Prolog – 4 Exercise

Which of the following are syntactically correct facts (indicate yes=correct, no=incorrect)

- Hazlenuts.
- tomsRedCar.
- 2Ideas.
- Prolog.

Proposition Logic in Prolog – 5 Exercise

Given the database below, study the queries below it. Again indicate whether you think the goal will succeed or not by answering *yes* or *no*.

```
blue_box.
red_box.
green_circle.
blue_circle.
orange_triangle.

?- green_circle.
?- circle_green.
?- red_triangle.
?- red_box.
?- orange_Triangle.
```

FOL in Prolog

More complicated facts consist of a relation and the items that this refers to. These items are called arguments. Facts can have arbitrary number of arguments from zero upwards. A general model is shown below:

```
relation(<argument1>,<argument2>,....,<argumentN> ).
```

Relation names must begin with a lowercase letter

```
likes(john,mary).
```

FOL in Prolog - 2

An example database. It details who eats what in some world model.

```
eats(fred,oranges).           /* "Fred eats oranges" */
eats(fred,t_bone_steaks). /* "Fred eats T-bone steaks" */
eats(tony,apples).            /* "Tony eats apples" */
eats(john,apples).            /* "John eats apples" */
eats(john,grapefruit).       /* "John eats grapefruit" */
```

FOL in Prolog - 3

If we now ask some queries we would get the following interaction:

```
?- eats(fred,oranges).        /* does this match anything in the database? */
yes                       /* yes, matches the first clause in the database */

?- eats(john,apples).         /*  do we have a fact that says john eats apples? */
yes                        /* yes we do, clause 4 of our eats database */

?- eats(mike,apples).         /* how about this query, does mike eat apples */
no                         /* not according to the above database. */

?- eats(fred,apples).         /* does fred eat apples */
no                         /* again no, we don't know whether fred eats apples */
```

FOL in Prolog - 3

Let us consider another database. This time we will use the predicate age to indicate the ages of various individuals.

```
age(john,32).              /*  John is 32 years old */
age(agnes,41).              /*  Agnes is 41 */
age(george,72).             /*  George is 72 */
age(ian,2).                /*  Ian is 2 */
age(thomas,25).             /*  Thomas is 25 */
```

If we now ask some queries we would get the following interaction:

```
?- age(ian,2).    /* is Ian 2 years old? */
yes                  /* yes, matches against the fourth clause of age */

?- agnes(41).    /* for some relation agnes are they 41  */
no                    /* No.   In the database above we only know about the relation */
                      /*  age, not about the relation agnes, so the query fails */

?- age(ian,two)  /* is Ian  two years old? */
no                     /* No.  two and 2 are not the same and therefore don't match */
```

Variables and Unification

How do we say something like "What does Fred eat"? Suppose we had the following fact in our database:

eats(fred,mangoes).

How do we ask what fred eats. We could type in something like

?- eats(fred,what).

However Prolog will say no. The reason for this is that what does not match with mangoes. In order to match arguments in this way we must use a **Variable**.

The process of matching items with variables is known as unification. Variables are distinguished by starting with a capital letter.

Variables and Unification - 2

```
X           /* a capital letter */
VaRiAbLe    /* a word - it be made up or either case of letters */
My_name     /* we can link words together via '_' (underscore) */
```

Thus returning to our first question we can find out what fred eats by typing

```
?- eats(fred,What).

What=mangoes

yes
```

Variables and Unification – 3

Let's consider some examples using facts. First consider the following database.

```
loves(john,mary).
loves(fred,hobbies).
```

Now let's look at some simple queries using variables

```
?- loves(john,Who).     /* Who does john love? */
Who=mary                /* yes , Who gets bound to mary */
yes                     /*  and the query succeeds*/


?- loves(arnold,Who)    /* does arnold love anybody */
no                      /* no,  arnold doesn't match john or fred */


?- loves(fred,Who).     /* Who does fred love */
Who = hobbies           /*  Note the to Prolog Who is just the name of a variable, it */
yes                     /* semantic connotations are not picked up, hence  Who unifies */
                        /* with hobbies */
```

Variables and Unification - 4

```
tape(1,van_morrison,astral_weeks,madam_george).
tape(2,beatles,sgt_pepper,a_day_in_the_life).
tape(3,beatles,abbey_road,something).
tape(4,rolling_stones,sticky_fingers,brown_sugar).
tape(5,eagles,hotel_california,new_kid_in_town).
```

Let's now look at some queries.

```
?- tape(5,Artist,Album,Fave_Song).          /* what are the contents of tape 5 */
Artist=eagles
Album=hotel_california
Fave_Song=new_kid_in_town

yes

?- tape(4,rolling_stones,sticky_fingers,Song).     /* find just  song */
Song=brown_sugar                                   /* which you like best from the album */
yes
```

Variables and Unification – 5 Exercise

Which of the following arguments will unify?

eats(fred,tomatoes) *? eats(Whom,What)*	likes(jane,X) *? likes(X,jim)*
eats(fred,Food) *? eats(Person,jim)*	f(X,Y) *? f(P,P)*
cd(29,beatles,sgt_pepper). *? cd(A,B,help).*	f(foo,L) *? f(A1,A1)*
f(X,a) *? f(a,X)*	

Rules - 1

Consider the following

'All men are mortal':

We can express this as the following Prolog rule

mortal(X) :- human(X).

The clause can be read in two ways (called either a declarative or a procedural interpretation).

- The declarative interpretation is "For a given X, X is mortal if X is human."
- The procedural interpretation is "To prove the main goal that X is mortal, prove the subgoal that X is human."

Rules - 2

To continue our previous example, lets us define the fact 'Socrates is human' so that our program now looks as follows:

```
mortal(X) :- human(X).
human(socrates).
```

If we now pose the question to Prolog

```
?- mortal(socrates).
```

The Prolog interpreter would respond as follows:

yes

Rules - 3

We can also use variables within queries. For example, we might wish to see if there is somebody who is mortal. This is done by the following line.

```
?- mortal(P).
```

The Prolog interpreter responds.

```
P = socrates

yes
```

Rules - 4

Sometimes we may wish to specify alternative ways of proving a particular thing. This we can do by using different rules and facts with the same name. For example, we can represent the sentence 'Something is fun if its a red car or a blue bike or it is ice cream' as follows:

```
fun(X) :-          /* an item is fun if */
    red(X),        /* the item is red */
    car(X).        /* and it is a car */


fun(X) :-          /*  or an item is fun if */
    blue(X),       /* the item is blue */
    bike(X).       /* and it is a bike */

fun(ice_cream).        /* and ice cream is also fun. */
```

Rules - 5

```
fun(X) :- red(X), car(X).
fun(X) :- blue(X), bike(X).
```

Looks to Prolog Like

```
fun(X_1) :- red(X_1), car(X_1).
fun(X_2) :- blue(X_2), bike(X_2).
```

Thus variable name scoping is per-individual rule (often called a clause). The same variable name may appear in different clauses of a rule, or rules with different names. Each time it is treated as something specific to its context. A variable X may occur in the program many times with many different bindings. It will only have the same bindings when you tell it to.

Rules - 6

Consider the following program:

```
fun(X) :- red(X), car(X).
fun(X) :- blue(X), bike(X).
car(vw_beatle).
car(ford_escort).
bike(harley_davidson).
red(vw_beatle).
red(ford_escort).
blue(harley_davidson).
```

Let's now use the above program and see if a harley_davidson is fun. To do this we can ask Prolog the following question.

```
?- fun(harley_davidson).        /* to which Prolog will reply */
yes                             /* to show the program succeeded */
```

Rules - 7

```
fun(X) :- red(X), car(X).
fun(X) :- blue(X), bike(X).
car(vw_beatle).
car(ford_escort).
bike(harley_davidson).
red(vw_beatle).
red(ford_escort).
blue(harley_davidson).
```

We can also ask our program to find fun items for us. To do this we can pose the following question.

```
?- fun(What).
```

To which Prolog will reply

```
What=vw_beatle
```

yes

Rules – Exercise 1

Which of these are syntactically correct?

- a :- b, c, d:- e f.
- happy(X):- a , b.
- happy(X):- hasmoney(X) & has_friends(X).
- fun(fish):- blue(betty), bike(yamaha).

Rules – Exercise 2

Given the database below, study the queries underneath it. Indicate whether you think a particular query will succeed or fail by answer yes or no using the buttons.

```
likes(john,mary).
likes(john,trains).
likes(peter,fast_cars).
likes(Person1,Person2):-
      hobby(Person1,Hobby),
      hobby(Person2,Hobby).
hobby(john,trainspotting).
hobby(tim,sailing).
hobby(helen,trainspotting).
hobby(simon,sailing).
```

Which of the following will succeed:

?- likes(john,trains).

?- likes(helen,john).

?- likes(tim,helen).

?- likes(john,helen).

Search: Backtracking

Suppose that we have the following database

```
eats(fred,pears).
eats(fred,t_bone_steak).
eats(fred,apples).
```

So far we have only been able to ask if fred eats specific things. Suppose that I wish to instead answer the question, "What are all the things that fred eats". To answer this I can use variables again. Thus I can type in the query

?- eats(fred,FoodItem).

As we have seen earlier, Prolog will answer with

FoodItem = pears

This is because it has found the first clause in the database. At this point Prolog allows us to ask if there are other possible solutions. When we do so we get the following.

FoodItem = t_bone_steak

if I ask for another solution, Prolog will then give us.

FoodItem = apples

Backtracking in Rules

We can also have backtracking in rules. For example consider the following program.

```
hold_party(X) :- birthday(X), happy(X).
birthday(tom).
birthday(fred).
birthday(helen).
happy(mary).
happy(jane).
happy(helen).
```

If we now pose the query

?- hold_party(Who).

Search Example

Consider the following knowledgebase

```
fun(X) :- red(X),  car(X).
fun(X) :- blue(X), bike(X).

red(apple_1).
red(block_1).
red(car_27).

car(desoto_48).
car(edsel_57).

blue(flower_3).
blue(glass_9).
blue(honda_81).
bike(iris_8).
bike(my_bike).
bike(honda_81).
```

Queries:

?- fun(What).

Recursion

"we often wish to repeatedly perform some operation either over a whole data-structure, or until a certain point is reached"

This simply means a program calls itself typically until some final point is reached.

Recursion Example

```
parent(john,paul).          /* paul is john's parent */
parent(paul,tom).           /* tom is paul's parent */
parent(tom,mary).            /* mary is tom's parent */

ancestor(X,Y):- parent(X,Y).   /* someone is your ancestor if there are your parent */

ancestor(X,Y):- parent(X,Z),   /* or somebody is your ancestor if they are the parent */
                ancestor(Z,Y). /* of someone who is your ancestor */
```

The above program finds ancestors, by trying to link up people according to the database of parents at the top to the card. So let's try it out by asking

```
?- ancestor(john,tom).
```

Recursion Exercises

```
a(X):- b(X,Y), a(X).

go_home(no_12).
go_home(X):- get_next_house(X,Y), home(Y).

foo(X):- bar(X).

lonely(X):- no_friends(X).
no_friends(X):- totally_barmy(X).

has_flu(rebecca).
has_flu(john).
has_flu(X):- kisses(X,Y),has_flu(Y).
kisses(janet,john).

search(end).
search(X):- path(X,Y), search(Y).
```

Recursion Exercises

town1----->-----town2---->----town3---->----town4--->----town5---->---town6

A one way road links 6 towns. Write a program that can work out if you can travel on that road. For example. Here are two sample program behaviors.

?- can_get(town2,town5).
yes

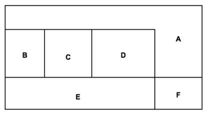
?- can_get(town3,town1).
no

Last Week’s Assignment

Encode Knowledgebase:

```
adjacent(a,b).
adjacent(a,c).
adjacent(a,d).
adjacent(a,f).
adjacent(b,c).
adjacent(b,e).
adjacent(c,d).
adjacent(c,e).
adjacent(d,e).
adjacent(e,f).

color(red).
color(blue).
color(green).
color(yellow).
```



Encode constraints:

```
assign(A, X, B, Y) :-
   color(X), color(Y), adjacent(A,B), X\=Y.

solution(A, B, C, D, E, F, G, AC, BC, CC, DC, EC, FC, GC) :-
   assign(A,AC,B,BC),
   assign(A,AC,C,CC),
   assign(A,AC,D,DC),
   assign(A,AC,F,FC),
   assign(B,BC,C,CC),
   assign(B,BC,E,EC),
   assign(C,CC,D,DC),
   assign(C,CC,E,EC),
   assign(D,DC,E,EC),
   assign(E,EC,F,FC).
```