CHAPTER

25

PROLOG – THE NATURAL LANGUAGE OF ARTIFICIAL INTELLIGENCE

Sophism: A Cretan saying all Cretans are liars!

25.1 INTRODUCTION

Learning AI should never be a theoretical exercise. It has to be augmented by programs that actually exhibit it. Naturally one requires a programming language for this. While numerous languages (such as BASIC, C and Java) have been used to write AI code, the better known languages are PROLOG and LISP. PROLOG was first developed by Alain Colmerauer of the University of Marseilles, France in the early 1970s and was used as a tool for Programming in Logic. PROLOG is known for its in-built depth-first search engine and for its ease in quick prototyping. A PROLOG program, unlike other computer language programs, comprises a description of the problem using a number of *facts* and *rules*. Execution is more of defining a goal that forces the PROLOG Inference Engine to search and move in a direction that yields one or more solutions that help achieve the goal.

Section 6.2 gave a brief description about PROLOG from the point of view of logic. In this chapter we try to cover the basic principles of programming in PROLOG to enable the reader to comprehend and program for AI. It may be noted that only those features that are prominent and essential to arm the reader to write AI programs are discussed. As will be seen the language is simple, more akin to English and thus quickly learnt. A variety of PROLOG compilers are currently available. The programs discussed herein have been written and compiled using LPA WIN-PROLOG. Most of the programs do not use LPA WIN-PROLOG specific predicates and thus will work fine on other PROLOG environments as well; for all others you will require the LPA WIN-PROLOG compiler.

25.2 CONVERTING ENGLISH TO PROLOG FACTS AND RULES

It is extremely simple to convert English sentences or facts into their PROLOG equivalents. Here are a few sentences converted into PROLOG.

English	**PROLOG**
The cakes are delicious.	delicious(cakes).
The pickles are delicious.	delicious(pickles).
Biryani is delicious.	delicious(biryani).
The pickles are spicy.	spicy(pickles).
Priya relishes coffee.	relishes(priya,coffee).
Priya likes food if they are delicious.	likes(priya, Food) if delicious(Food).
Prakash likes food if they are spicy and delicious.	likes(prakash, Food) if spicy(Food) and delicious(Food).

While the first four PROLOG equivalents are called *Facts* the last two conversions are termed as *Rules*. *Food* is a variable whose value is to be found from the previous facts. Also note that while variables begin with an upper case letters, the constants (eg. *priya*, *pickles*, etc.) begin with lower case letters. If we just look at the contents of the second column, you will be amazed to note that we have already written a PROLOG program to deduce what *Priya* and *Prakash* like! So here is how our first PROLOG program looks like-

```
/* This is a program that can be used to find
   what a person likes*/
delicious(cakes).               % Cakes are delicious.
delicious(pickles).             % Pickles are delicious.
delicious(biryani).             % Biryani is delicious.
spicy(pickles).                 %Pickles are spicy.
relishes(priya,coffee).         %Priya relishes coffee.
likes(priya, Food):-
        delicious(Food).        % Priya likes food if it is delicious.
likes(prakash, Food) :-
        spicy(Food), delicious(Food). /*Prakash likes food if it is spicy and delicious.*/
```

Notice how the **ifs** have been replaced by (:-) and the **and**s by (commas). Also every fact such as spicy(pickles) or rule is terminated by a . (period). Comments could be of any of the following forms

```
% This comments only this line.
/* This comments everything in between the asterisks and the slashes.*/
```

25.3 GOALS

Now compile the program. The program is ready to accept *goals* which are more like queries. Suppose we wish to ask the program-

Which food items are delicious?

This, in PROLOG terminology, is called a *Goal* and is presented on the **?** – prompt as

```
?- delicious(Food).
Food = cakes
Press ; to get the remaining alternatives for Food viz.
Food = pickles;
Food = biryani
```

Note that the goal also ends with a . (period).
How do we find out the kind of food a person likes. Try the goal:

likes(priya,Food).

which means "*What food does Priya like?*".
The goal and the response of the system is shown below:

```
| ?- likes(priya,Food).
Food = cakes
Press ; to get the remaining alternatives for Food viz.
Food = cakes ;
Food = pickles ;
Food = biryani
| ?-
```

Now try the goal:
likes(prakash,Food).

The above program serves several goals. Apart from finding out what food *Priya* and *Prakash* like, you can also find which food items are spicy and who relishes what. More complex goals could also be presented. For instance you could ask other questions like:

Who relishes coffee and also likes pickles?

using the goal given below.

```
?- relishes(Who,coffee),likes(Who,pickles).
Who = priya
```

Alternatively, you could also query the likings of a person who relishes coffee as:

```
?- relishes(Who,coffee), likes(Who,Food).
```

for which you will get the following answers:

```
Who = priya ,
Food = cakes ;

Who = priya ,
Food = pickles ;

Who = priya ,
Food = biryani
```

Now do you like PROLOG or not? If the answer is negative then try writing code in any of the other conventional languages (C, BASIC, Java,...) to realize the same! Don't forget that the same PROLOG program can still reason around and satisfy many other goals. Try these as goals *delicious(cakes).*

meaning - Are cakes delicious?
and
spicy(pickles).
Are pickles spicy?
likes(priya,biryani).
Does Priya like biryani?

All these goals will prompt the PROLOG inference engine to yield a *Yes* as all of them are *True*.
By now you are already familiar with what *facts* and *rules* are and how *goals* serve to query information. You are also aware of *Variables* and *Constants* in PROLOG.

25.4 PROLOG TERMINOLOGY

(a) Predicates

A *predicate* name is the symbol used to define a relation. For instance in relishes(priya, coffee). the symbol *relishes* is the predicate while the contents within viz. *priya* and *coffee* comprise its arguments. Predicates need not necessarily have arguments.

(b) Clauses

Clauses are the actual rules and facts that constitute the PROLOG program.

(c) Atoms

Atoms are basically symbols or names that are indivisible and are used in the program, files or as database items, among others. They are represented in single quotes as in
'Here is an atom'

(d) Character

Character lists are used in programs that manipulate text character by character. They are represented in double quotes and when printed appear as a list of ASCII codes of each letter in the character chain.

(e) Strings

Strings on the contrary, have a form that is in between an atom and a character list. They are represented between backward quotes (the key below *esc* on the keyboard) as –
`This is an string`.

(f) Arity

The number of arguments in a predicate forms its arity. It is represented by a */n* placed after the predicate name. *n* is the number of arguments. For instance the predicate *relishes/2* takes two arguments.

25.5 VARIABLES

In PROLOG, variables must begin with a capital letter and could be followed by other letters (upper or lower case), digits, underscores or hyphenations but no blanks. Examples of a few valid variables are given below:

My_favourite-food_items
Menu-Item-1
Priya
Does_Priya_like_Coffee

Some invalid variables are given below:

Menu Item-1
5Pro
foo

PROLOG also has a variable called the *anonymous* variable. This is represented by just an underscore (_). In a goal if information on a particular argument is not required then an anonymous variable could be used in its place. If you need to know only the color of a car and not its color from a program containing facts of the type car(Brand,Color) viz.

car(maruti,white).
car(fiat,black).
then the goal –
car(_,Color).

will provide only the colors of the cars available in the fact base and not the car names. In short the goal could be translated into the English query - Find only the color(s) of the car(s).

25.6 CONTROL STRUCTURES

In PROLOG it is possible to order a set of rules in two different ways. Examine the flow in the following program:

```
p:- a,b,c.
p:- d,e,f.
```

This means that *p* is true if either (*a and b and c*) is true or (*d and e and f*) is true. The comma as mentioned earlier implies an *and*. The same rules could be reordered as –

```
p:- a,b,c;d,e,f.
```

without changing the logical meaning.

PROLOG also allows an *if then* control structure. The -> and ; together form the *if_then* control structure. Thus in

```
a:- b -> c;d.
```

the right hand side of the rule is interpreted as

If b is true then verify the truth of c else find that of d.

25.7 ARITHMETIC OPERATORS

Some of the main arithmetic related predicates are shown in Table 25.1.

Table 25.1 *Some Arithmetic Operators in PROLOG*

Predicate	*Functionality*
</2, >/2	Expression *less than, greater than*
= : =/2, =\=/2	Expression *equality, inequality*
is/2	Expression evaluator
seed/1	Seed the random number generator

An arithmetic expression can take any of the following forms:

(i) A number – could be an integer or a floating point. (E.g. 45)

(ii) A list wherein the elements are numbers. (E.g. [55, 45, 40])

(iii) A function represented by a compound term wherein the functor denotes a function predefined by the compiler and whose argument is itself an expression. (E.g. tan(1 + 4)/4)

(iv) A bracketed expression (An expression in a bracket) (E.g. (4**W*))

(v) A variable that has been bound to an expression before the expression is evaluated. (E.g. *V* where *V* was initialized to *V* = sin(*CX*))

Some of the basic predefined arithmetic, trigonometric and logarithmic functions are shown in Table 25.2.

Table 25.2 *Some predefined Arithmetic, Trigonometric and Logarithmic functions*

Function	*Computes*
Arithmetic	
X + Y	Sum
X – Y	Difference
X*Y	Product
X/Y	Division
–X	Negative
X mod Y	Remainder after integer division
X^Y	X to the power Y
Trigonometric	
rand(X)	Pseudo random floating point number between 0 and X
sqrt(X)	Square root of X
sin(X), cos(X), tan(X), asin(X), acos(X), atan(X)	Sine, Cosine and Tangent of X as also arc values respectively.
Logical	
X«Y, X»Y	Logical shift arithmetic left or right respectively
\(X)	Logical negation of the integer X
a(X,Y), o(X,Y), x(X,Y)	Logical AND,OR,XOR of X and Y respectively

25.8 MATCHING IN PROLOG

How does PROLOG find a way to satisfy the goal or query posed to it? This depends on the goal. If the goal does not have any variables as in

```
relishes(priya,coffee).
```

then things are simple. A matching is performed predicate to predicate and argument to argument. If this goal matches any fact then it outputs a *Yes* (or *No*) as the case may be.

Things are a bit different for a case when the goal involves variables. Consider the simple case of "*Who relishes coffee?*" The corresponding goal is

```
relishes(Who,coffee).
```

Remember *Who* is a variable. This goal is matched with the first fact of *relishes* viz. *relishes(priya,coffee)* which causes the variable *Who* to be bound to the value *priya* which in turn is finally returned.

Now add the following facts to the above program:

```
fond(priya,driving).
fond(prakash,cartoons).
```

Compound goals like – Who relishes coffee and is also fond of driving? viz.

```
relishes(Who, coffee), fond(Who,driving).
```

will initially bind the first *Who* to *priya* and then try to find a match with this value for the second goal viz. *fond(priya,driving)*.

Notice that *Who* has been bound to *priya*.

Since it could consistently find a value for *Who* that matches both the goals, the system returns *priya* as an answer.

The value bound in the first goal is thus propagated to the next goal and so on.

What about the goal

```
relishes(Who, coffee), fond(Who,cartoons). ?
```

Since the propagated value of *Who*, viz. *priya*, cannot bind to the variable *Who* in the second clause (obviously), the PROLOG inference engine *backtracks* to another fact for relishes to find a new value for *Who* that matches the first goal.

If you add the fact

```
relishes(prakash,coffee).
```

then this compound goal given above would in the second attempt, bind the variable *Who* to *prakash* and carry it forward to the second goal to finally satisfy it.

One should bear in mind that free variables having the same position in a predicate can bind to each other. For instance in the goal-

likes(priya,X).

the variable *X* is not bound (free) and hence will bind with the left hand side of the first rule viz.

likes(priya,Food).

which also has a variable as its second argument. We could thus say that *X* is mapped to the variable *Food* whose value is subsequently discovered if all the subgoals in the right hand side of the rule can are satisfied.

In brief one should bear in mind that PROLOG allows structures that comprise a name (or it could be an atom) with arguments enclosed in brackets. Two such terms *unify* only if their structure names, also called the functors, are same, the number of arguments within are same and every argument within it *unifies* with the corresponding one of the other.

25.9 BACKTRACKING

By now we have partially discussed what backtracking means. But let's finish it for good. Imagine you have entered a maze and are trying to search for something within. You would in the normal course always take the left turn (or right) consistently at every fork and continue. If you reach a dead end, you would return to the fork and try the right turn (or the left). Eventually you would search the whole maze. We have seen that in compound goals, PROLOG attempts to do the very same. The process is called *backtracking*. Whenever there is a fork or alternate paths to be discovered a backtracking point is put up and the same is visited in the event of a failure. Let's look at the following program.

```
likes(prakash, X) :-edible(X), tastes(X, sweet).
tastes(chocolates,sweet).
tastes(gourd,bitter).
tastes(toffees, sweet).
edible(chocolates).
edible(toffees).
edible(gourd).
```

What happens when you pose the goal:

likes(prakash,X).

The right hand side of the rule for *likes* is first triggered. So we have *edible(X)* interpreted as

Find an X such that X is edible

to be satisfied. There are three values that *X* could take (*chocolates*, *toffees* and *gourd*). The variable *X* is first bound to *chocolates*. The process of verifying whether chocolates are sweet continues using *tastes(chocolates, sweet)*. Since this too is true the system returns *chocolates* as an answer. If you press ; you can get further solutions for *X*.

Now, inspect the modified program below:

```
likes(prakash, X) :-edible(X), tastes(X, sweet),write(X),nl,fail.
tastes(chocolates,sweet).
tastes(gourd,bitter).
tastes(toffees, sweet).
edible(chocolates).
edible(toffees).
edible(gourd).
```

Note that we have added a new term called *fail* which effectively returns all possible answers one after the other. After the value of *X* has been bound to *chocolates* (as in the previous explanation) the next thing it encounters is a *write(X)* which writes the currently bound value of *X*. The *nl* forces a newline. Finally *fail* forces the PROLOG engine to imagine that the right hand side is not *true* though all previous clauses have succeeded in finding an apt value for *X*. *fail* thus makes the system feel that it has not succeeded in its attempt to satisfy a goal and therefore forces it to backtrack to the previous fork and find the next alternative solution for *X*. *fail* can be used when one needs to perform an exhaustive search of a tree and not just deliver a single instance of the correct solution. In other words the goal succeeds in finding all possible solutions by failing each time!

Thus, when we issue the goal:

```
| ?- likes(prakash, X).
```

here is what you get

```
chocolates
toffees
no
```

The final *no* crops up because after all solutions (two in this case) are found, the last search for another solution fails due to the *fail*!

This can be avoided by allowing an automatic success after the final failure of *likes*. Insert an additional terminating clause for *likes* as shown below.

```
likes(prakash, X) :-edible(X), tastes(X, sweet),write(X),nl,fail.
likes(_,_).          /*Terminating condition*/
tastes(chocolates,sweet).
tastes(gourd,bitter).
tastes(toffees, sweet).
edible(chocolates).
edible(toffees).
edible(gourd).
```

After all possible values of X have been found the second clause is triggered which in plain English states that *Anyone likes anyone*. Now try to interpret what would happen if this clause were to be placed prior to the original clause for *likes*.

Another way around is to use the following program.

```
do:-likes(prakash,X).
do.
likes(prakash, X) :-edible(X), tastes(X, sweet),write(X),nl,fail.
tastes(chocolates,sweet).
tastes(gourd,bitter).
tastes(toffees, sweet).
edible(chocolates).
edible(toffees).
edible(gourd).
```

The readers are urged to find the difference between these two programs.

25.10 CUTS

Written using an exclamatory mark (!), cuts form a way of preventing backtracking. Imagine what would happen if we burn the bridge on a river after crossing it. It would not be possible for us to return to find an alternate path, if required. The cut does just that. Inspect the sequence in which the subgoals on the right hand side are executed in following program-

```
do:- p(N),q(N),nl,write(N),!,r(N).
p(1).
p(2).
p(3).
p(4).
q(2).
q(3).
q(4).
r(3).
```

When the goal *do* is issued, the inference engine tries to satisfy *p(N)*; then *q(N)* after which it proceeds to satisfy *r(N)* which happens to be located after the cut (!). Naturally, after satisfying *p(N)* (meaning finding a value of *N* such that *p(N)* is true), if *q(N)* is not satisfied, the engine backtracks to another alternative for *p* and comes back to check whether *q* can be satisfied with the new state of *p*. But if *r* (or any alternatives of *r*) is not satisfied, given the currently satisfied states of *p* and *q*, *do* fails as the cut (!) prevents any further backtracking. In short once *p* and *q* are satisfied, it becomes mandatory that *r* be satisfied, else the rule *do* fails. The sequence below shows how *N* changes its values.

```
p(1) → q(1) False
          ← Backtrack to p
p(2) → q(2)
!      → r(2) False
       Cannot backtrack; thus fails.
```

What would happen if there is an alternative path for *do* as in -

```
do:- p(N),q(N),nl,write(N),!,r(N).
do.
```

The effect would remain the same, thanks to the cut that makes the inference engine forget about the fork at *do* in toto! Now try adding the fact *r(2)* and see the effect.

Here is a simple program that helps select a car described by its brand name, color and cost.

```
car(palio,black, 450000).
car(hyundai, silver, 300000).
car(fiat, silver, 400000).
car(maruti, black, 20000).

get_car(Color,Cost_Less_Than):-
          car(Name, Color,Cost),!,Cost<Cost_Less_Than, nl, write(Name).
```

The goal

```
get_car(black,350000).
```

does not yield a solution in spite of the fact that one of the cars viz. the *maruti* satisfies the same. This is so because after the variables - *Name* and *Cost* are bound to *palio* and *450000*, the cut is crossed and the condition *350000 < 450000* is checked. Since the condition is not satisfied a backtracking is initiated but prevented due to the cut, yielding no solutions. Now re-order the facts for car with *car(maruti, black, 20000)* as the very first one and try the same goal.

25.11 RECURSION

A recursive procedure is one that calls itself. A simple recursive procedure often used for implementing loops, is given below –

```
repeat.
repeat:- repeat.
```

Such a program becomes handy when we have only one solution and we need to introduce backtracking. The repeat structure makes PROLOG's inference engine to believe that there are infinite number of solutions. Inspect the next program which will make things clearer.

```
getinput:-
redo,
grab(Ch),string_chars(C,[Ch]),
write(C),Ch= =13.
redo.
redo:- redo.
```

The *grab/1* predicate takes in a single character in ASCII directly from the keyboard while the *string_chars/2* predicate converts the input into a regular string which is displayed by *write*. The second argument of the latter predicate has to be a list which is why the variable *Ch* is presented as a list, a structure discussed in the next section. Both *grab* and *string_chars* are LPA PROLOG specific. You can always find the equivalent of these predicates from the documentations of the PROLOG compiler you are using.

The more important issue here is the role played by the predicate *redo*. As is obvious, the clauses for *redo* form a recursive loop. If you try to issue the goal as-