# Chapter 14: Introduction to prolog

## 14.1 Converting English to Prolog:

Here are 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):-delicious(Food). |
| Prakash likes food if they are spicy and delicious. | likes(prakash,Food):-spicy(Food),delicious(Food). |

In PROLOG:
Ifs have been replaced by (:-).
Commas have been replaced by (,).
Rule and Goal is terminated by a (.).
Comments could be of any of the following forms:
%this comments only this line.
/* this comments everything in between the asterisks and the slashes. */

## 14.2 Goals:

Now compile the program.

- Suppose we wish to ask the program:
  "Which food items are delicious?"
- This, in PROLOG terminology, is called 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
```

- Try the goal:

```
?-likes (priya , Food).
Food = cakes;
Food = pickles;
Food = biryani
```

- You could ask other questions like:
- Who relishes coffee and also likes pickles?
- Using goal given below:

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

## 14.3 Prolog Terminology

### 1. Predicates:

A predicate name is the symbol used to define a relation.

For instance: relishes(priya , coffee) where relishes is predicate while the contents within viz. priya and coffee comprise its arguments.

### 2. Clauses:

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

### 3. 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'.

### 4. Character:

Character list are used in programs that manipulate text character by character. They are represented in double quotes.

### 5. Strings:

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

### 6. **Arity**:

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

## 14.4 Variables

- In PROLOG, variables must begin with a capital letter and could be followed by other letters (upper or lower case), underscores or hyphenations but no blanks.

  Ex:

  My_fav_food_items

  Menu-item-1

  Priya

  Priya_Coffee

## 14.5 Backtracking / Fail:

- Consider the following program: (without using fail)

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

Goal:
?- likes(prakash,X).
chocolates
```

- Now inspect the modified program below: (using fail)

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

Goal:
?- likes(prakash, X).
chocolates
toffees
false
```

- We have added a new item called **fail** which effectively returns all possible answers one after the other. After the value of X has been bound to chocolates 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 through 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 alternative solution for X.

- The final false crops because after all solutions are found, the last search for another solution fails due to the fail.

- This can be avoided by inserting an additional terminating clause for likes as shown below.

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

Goal:

```
?- likes(prakash, X).
chocolates
toffees
true
```

## 14.6 Cuts

- Written using an exclamatory mark (!), cuts from a way of preventing backtracking.

- Without using cut:

```
car(etios,black,450000).
car(i10,silver,300000).
car(i20,silver,400000).
```

```
car(swift,black,200000).
getcar(Color,Costlessthan):-
car(Name,Color,Cost),Cost<Costlessthan,nl,write(Name).

Goal:
?- getcar(black,250000).
swift
```

- With using cut:

```
car(etios,black,450000).
car(i10,silver,300000).
car(i20,silver,400000).
car(swift,black,200000).
getcar(Color,Costlessthan):-
car(Name,Color,Cost),!,Cost<Costlessthan,nl,write(Name).

goal:
?- getcar(black,250000).
False
```

## 14.7 Recursion OR
## Demonstrate the use of Repeat Predicate in Prolog with example.

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

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

- **Ex: Factorial of a number**

```
factorial(0,X):- X=1.
factorial(N,Fact_N):-N>0,Q is N-1,factorial(Q,Fact_Q), Fact_N is N* Fact_Q.

goal:
?- factorial(4,X).
```

X=24.

## 14.8 Lists

- Lists in PROLOG constitute elements separated by commas and enclosed within square brackets. The elements could comprise of any data type. A list of integers looks like this:

  [2, 4, 6, 8]

- While that of string is given below:

  [`jan`, `feb`, `march`]

- A list comprises of two parts:
  1) Head: the first element of the list.
  2) Tail: the list comprising of all other members.

  For instance: the list [2,4,6,8] has the integer 2 as its head and [4,6,8] as its tail.

- **Member of a list:**

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

  The first clause states: X is member of the list if X is the Head o that list.

  The second clause states: else check to see if X is the member of the tail of the list.

- **Append a list to another:**

  The clauses for append are given below:

  ```
  append([],L,L).
  append([X|L1],L2,[X|L3]):-append(L1,L2,L3).
  ```

The first clause states: if the list to be appended is an empty list then output the other list as the appended one.

The second clause states: it takes the head of the list X to be appended and adds it to the head of the output list, ([X|L3]), and calls append recursively with the tail of the list to be appended, the list to which it is to be appended and L3 as its arguments. Note that L3,the tail of output list gets values only while the unwinding of the recursive loop occurs.

## ❖ Prolog Programs

**QB-1/10/13) Demonstrate the use of cut and fail predicates in Prolog with example.**
**Solution:** refer 14.5 and 14.6

**QB-2) Write a Prolog program to solve Tower of Hanoi problem.**
**Solution:**

```
move(1,X,Y,_):-write('Move disk from '),write(X),write(' to '),write(Y),nl.
move(N,X,Y,Z):-N>1,M is N-1,
move(M,X,Z,Y),
move(1,X,Y,_),
move(M,Z,Y,X).
```

**Goal:** ?- move(3,x,y,z).

```
Move disk from x to y
Move disk from x to z
Move disk from y to z
Move disk from x to y
Move disk from z to x
Move disk from z to y
Move disk from x to y
true
```

**QB-3) Write a Prolog program to count vowels in a list of characters.**
**Solution:**

```
vowel(X):- member(X,[a,e,i,o,u]).
number_vowel([],0).
number_vowel([X|T],N):- vowel(X),number_vowel(T,N1), N is N1+1.
number_vowel([X|T],N):- number_vowel(T,N).
```

**Goal**: ?- number_vowel([k,i,n,j,a,l],N).
N = 2

**QB-4) Write a Prolog program to find sum of elements of a list.**

```
list_sum([],0).
list_sum([H|T],Total):-list_sum(T,Sum1),Total is H+Sum1.
```

**Goal**: ?- list_sum([1,2,8],Total).
Total = 11.

**QB—5) Demonstrate the use of Repeat Predicate in Prolog with example.**

**Solution:** refer 14.7

**QB-6) What is red cut and green cut in Prolog?**

**Solution:**

**Green cut**

A use of a cut which only improves efficiency is referred to as a green cut. Green cuts are used to make programs more efficient without changing program output. For example:

```
gamble(X) :- gotmoney(X),!.

gamble(X) :- gotcredit(X), \+ gotmoney(X).
```

This is called a green cut operator. The ! tells the interpreter to stop looking for alternatives; however, if gotmoney(X) fails it will check the second rule. Although checking for gotmoney(X) in the second rule may appear redundant since Prolog's appearance is dependent on gotmoney(X) failing before, otherwise the second rule would not be evaluated in the first place. Adding \+ gotmoney(X) guarantees that the second rule will always work, even if the first rule is removed by accident or changed, or moved after the second one.

**Red cut**

A cut that is not a *green cut* is referred to as a *red cut*, for example:

```
gamble(X) :- gotmoney(X),!.

gamble(X) :- gotcredit(X).
```

Proper placement of the cut operator and the order of the rules is required to determine their logical meaning. If for any reason the first rule is removed (e.g. by a cut-and-paste accident) or moved after the second one, the second rule will be broken, i.e., it will not guarantee the rule \+ gotmoney(X).

**QB-7) Write a Prolog program to find factorial of a given number.**

**Solution:** factorial(0,X):- X=1.
factorial(N,X):-N>0,Q is N-1,factorial(Q,XX),X is N*XX.

**Goal**: ?- factorial(4,X).
X = 24

**QB-8) Write prolog program to append List2 to List1 and bind the result to List3.**

**Solution:** append([],L,L).
append([X|L1],L2,[X|L3]):-append(L1,L2,L3).

**Goal**: append([1,2,3],[4,5],X).
X=[1,2,3,4,5].

**QB-9) Write following prolog programs:**

1. **To find the factorial of a positive integer number.**

**Solution:**factorial(0,X):- X=1.
factorial(N,Fact_N):-N>0,Q is N-1,factorial(Q,Fact_Q),
Fact_N is N* Fact_Q.

**Goal**:?- factorial(4,X).
X=24.

2. **To find the nth element of a given list.**

element_at(X,[X|_],1).
element_at(X,[_|L],N) :- N > 1, N1 is N - 1, element_at(X,L,N1).

**Goal**: ?- element_at(X,[a,b,c,d,e],3).
X = c .

**QB-11) Write a Prolog program to reverse a given list.**

**Solution:** reverse([],Z,Z).
reverse([H|T],Z,Acc) :- reverse(T,Z,[H|Acc]).

**Goal:** ?- reverse([a,b,c],X,[]).
X = [c, b, a].

**QB-12) Explain how list is used in Prolog. Discuss how following list functions can be implemented in Prolog.**

**(1) Checking membership of an item in a given list**

member(X,[X|_]).

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

**Goal**: ?- member(a,[a,b]).
True
?- member(1,[3,2]).
false.

**(2) Concatenating two lists**

```
append([],L,L).
append([X|L1],L2,[X|L3]):-append(L1,L2,L3).
```

**Goal:** ?- append([c],[a,b],L3).
L3 = [c, a, b].
?- append([c],[a,b],[a,b,c]).
false.
?- append([c],[a,b],[c,a,b]).
true.

**(3) Deleting an item in a given list.**

**Solution:**

```
del(X,[X|Tail],Tail).
del(X,[Y|Tail],[Y|Tail1]):- del(X,Tail,Tail1).
```

**Goal:** ?- del(a,[a,b],L3).
L3 = [b] .
?- del(b,[a,b],L3).
L3 = [a]

**QB-14) Write a program to find maximum number from a list.**

**Solution**:

```
max_l([X],X) .
max_l([H|T], M):- max_l(T, M), M >= H.
max_l([H|T], H):- max_l(T, M),H > M.
```

**Goal:** ?- max_l([7,2,1,6],X).
X = 7

**QB-15) Write a Prolog program for finding a set, which is result of the intersection of the two given sets.**

**Hint:** **Goal: intersect([1, 2, 3], [2, 3, 4], A)**
**A = [2, 3]**
**Goal: intersect([d, f, g], [a, b, c ], X)**
**X = []**

**Solution:** intersect([X|Y],M,[X|Z]) :- member(X,M), intersection(Y,M,Z).
intersect([X|Y],M,Z) :- \+ member(X,M), intersection(Y,M,Z).
intersect([],M,[]).

**Goal:** ?- intersect([1,2,3],[2,3],M).
M = [2, 3]

**QB-17) Write a Prolog program that verifies whether an input list is a palindrome.**
**Hint: Goal: palindrome([r,a,c,e,c,a,r])**
**Output: Yes**
**Goal: palindrome([a,b,c])**
**Output: No**
**Solution:** palindrome(L) :- reverse(L,L).

**Goal:** ?- palindrome([a,b,b,a]).
true.

?- palindrome([a,b,b,c]).
False.

**QB-18) Write a Prolog program to find: the last element and the nth element (where 'n' indicates position), of an input integer list.**
**Hint: Goal: last_element([1, 2, 3, 8, 9], X)**
**Output: X = 9**
**Goal: nth_element([1, 2, 3, 8, 9], 4, X) (Here, n = 4)**
**Output: X = 8**
**Solution:** my_last(X,[X]).
my_last(X,[_|L]) :- my_last(X,L).
element_at(X,[X|_],1).
element_at(X,[_|L],N) :- N > 1, N1 is N - 1, element_at(X,L,N1).

**Goal:** ?- my_last(X,[a,b]).
X = b .

?- element_at(X,[a,b,c,d,e],3).
X = c .

**QB-19) Explain following terms with reference to Prolog programming language: Caluses, Predicates, Domains, Goals, Cut, Fail, Inference engine.**
**Solution:**

## 1. Clauses:

Rules allow us to make conditional statements about our world. Each rule can have several variations, called **clauses**.
These clauses give us different choices about how to perform inference about our world. Let's take an example to make things clearer.
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."

## 2. Predicate Definitions

**Built-in Predicates**
Prolog has a large number of built-in predicates. The following is a partial list of predicates which should be present in all implementations.

**Input predicates**
read(X): Read one clause from the current input and unify it with X. If there is no further input, X is unified with end_of_file.

see(File): Open File as the current input file.

seen :Close the current input file.

**Output predicates**
write(X): Write the single value X to the current output file.

**Control predicates**
X ; Y : X or Y. Try X first; if it fails (possibly after being backtracked into), try Y.

(X -> Y) : If X, then try Y, otherwise fail. Y will not be backtracked into. fail Never succeed.

**Arithmetic predicates**
X + Y: *When evaluated,* yields the sum of X and Y.
X – Y: *When evaluated,* yields the difference of X and Y.

X * Y: *When evaluated,* yields the product of X and Y.
X / Y: *When evaluated,* yields the quotient of X and Y.

## 3. Domains

The (fd) in clp(fd) stands for *finite domain*. This domain could have millions of values, but it must be a finite list. We're only concerned with variables with a finite domain, the **finite domain** of the name. For our purposes that means we want to reason about domains that are sets of integers. You *do need* to give variables a domain before you try to label them! clp(fd) is a library included in the standard SWI-Prolog distribution. It solves problems that involve sets of variables, where relationships among the variables need satisfied. **For example**, if we don't know a person's actual height yet, we can still assert that it's more than 21 and less than 108 inches (shortest/tallest people known). When we eventually do find a solution, unreasonable values will be rejected.

```
:- use_module(library(clpfd)).
foo(X) :- X in 1..3 .
foo(X) :- X in 5..7 .
foo(X) :- X in 8..12 .
```

## 4. Goals

A query to the Prolog interpreter consists of one or more goals.

For example, in
?- lectures(john, Subject), studies(Student, Subject).
there are two goals, lectures(john, Subject) and studies(Student, Subject).

A goal is something that Prolog tries to *satisfy* by finding values of the variables (in this case Student and Subject) that make the goal succeed.

## 5. Cut

Refer 14.6

## 6. Fail

Refer 14.5

## 7. Inference engine

An **inference engine** is a tool from artificial intelligence. The first inference engines were components of expert systems. The typical expert system consisted of a knowledge base and an inference engine. The knowledge base stored facts about the world. The inference engine applies logical rules to the knowledge base and deduced new knowledge. This process would iterate as each new fact in the knowledge base could trigger additional rules in the inference engine.

Inference engines work primarily in one of two modes either special rule or facts: forward chaining and backward chaining.