**Prolog Tutorial-2**

Dr A Sahu  
Dept of Computer Science & Engineering  
IIT Guwahati

---

**Prolog Relational Database: Example**

\[ [a, b, c] = [a|[b,c]] \]

Relation **append** is a set of tuples of the form \( (X,Y,Z) \) where \( Z \) consist if \( X \) followed by the element of \( Y \).

<table>
<thead>
<tr>
<th>( X )</th>
<th>( Y )</th>
<th>( Z )</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>[]</td>
<td>[]</td>
</tr>
<tr>
<td>[a]</td>
<td>[]</td>
<td>[a]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>[a,b]</td>
<td>[c,d]</td>
<td>[a,b,c,d]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Relation are also called **predicates**.

Query: Is a given tuple in relation **append**?

?-append([a],[b],[a,b]).  yes

?-append([a],[b],[]).   no

---

**Writing append relation in prolog**

**Rules**

\[
\text{append}([],Y,Y).
\]

\[
\text{append}([H|X],Y,[H|Z]):-\text{append}(X,Y,Z).
\]

**Queries**

?-append([a,b],[c,d],[a,b,c,d]).  yes

?-append([a,b],[c,d],Z).  \( Z = \{a,b,c,d\} \)

?-append([a,b],Y,[a,b,c,d]).  \( Y = \{c,d\} \)

?-append([a,b],Y,[a,b,c,d]).  \( Z = \{c,d\} \)

---

**Prolog Program Structure**

/* At the Zoo */

elephant(gaj).
elephant(aswasthama).
panda(chi_chi).
panda(ming_ming).

dangerous(X) :- big_teeth(X).
dangerous(X) :- venomous(X).
guess(X, tiger) :- stripey(X), big_teeth(X), isaCat(X).
guess(X, zebra) :- stripey(X), isaHorse(X).

---

**Factorial Program**

factorial(0,1).

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

The Prolog goal to calculate the factorial of the number 3 responds with a value for \( W \), the goal variable:

?- factorial(3,W).

\( W = 6 \)
Unification

- Two terms unify
  - if substitutions can be made for any variables in the terms so that the terms are made identical.
  - If no such substitution exists, the terms do not unify.
- The Unification Algorithm proceeds by recursive descent of the two terms.
  - Constants unify if they are identical
  - Variables unify with any term, including other variables
  - Compound terms unify if their functors and components unify.

Unification Examples: 1
The terms \( f(X, a(b, c)) \) and \( f(d, a(Z, c)) \) unify.

\[
\begin{align*}
\text{X} &= d \\
\text{Z} &= b \\
\text{X} &= f(Y) \\
\text{Y} &= g(Y) \\
\text{X} &= f(Y) \\
\text{X} &= 1+2
\end{align*}
\]

The terms are made equal if \( d \) is substituted for \( X \), and \( b \) is substituted for \( Z \).
We also say \( X \) is instantiated to \( d \) and \( Z \) is instantiated to \( b \), or \( X/d, Z/b \).

A Brief Diversion into Arithmetic
The built-in predicate ‘is’ takes two arguments. It interprets its second as an arithmetic expression, and unifies it with the first. Also, ‘is’ is an infix operator.

\[
\begin{align*}
?- \text{X is } 2 + 2 * 2. \\
X &= 6 \\
?- 10 \text{ is } (2 * 0) + 2 \text{ } \text{<<} \text{ } 4. \\
\text{no} \\
?- 32 \text{ is } (2 * 0) + 2 \text{ } \text{<<} \text{ } 4. \\
\text{yes}
\end{align*}
\]
**is**

But ‘is’ cannot solve equations, so the expression must be ‘ground’ (contain no free variables).

?- Y is 2 * X.
   no

?- X is 15, Y is 2 * X.
   X = 15, Y = 30.

---

**A Brief Discussion into Anonymous Variables**

\[/* member(Term, List) */\]

\[member(X, [X|T]). \quad \text{Notice } T \text{ isn’t used}\]

\[member(X, [H|T]) :- member(X, T). \quad \text{Notice } H \text{ isn’t used}\]

\[member(X, [X|_]). \quad \text{ }\]

\[member(X, [_|T]) :- member(X, T). \quad \text{ }\]

---

**Length of a List**

\[/* length(List, Length) */\]

**Naïve method:**

\[\text{length([], 0).}\]

\[\text{length([H|T], N):-length(T, NT), N is NT+1.}\]

---

**Length of a List**

\[/* length(List, Length) */\]

**Tail-recursive method:**

\[\text{length(L, N) :- acc(L, 0, N).}\]

\[/* acc(List, Before, After) */\]

\[\text{acc([], A, A).}\]

\[\text{acc([H|T], A, N):- A1 is A+1,}\]

\[\text{acc(T, A1, N).}\]

---

**Sum of List and Sum of Square List**

\[Lsum([], 0).\]

\[Lsum([H | T], TSum) :-\]

\[\text{Lsum(T, Sum1), Tsum is H+ Sum1.}\]

\[Lsqsum([], 0).\]

\[Lsqsum([H | T], TSqSum) :-\]

\[\text{Lsqsum(T, Sum1), TsqSum is H*H + Sum1.}\]

---

**Length of a List**

?- length([apple, pear], N).
   N=2

?- length([alpha], 2).
   no

?- length(L, 3).
   L = [_, _, _]
   Yes
**List: Inner Product**

A list of n integers can be used to represent an n-vector (a point in n-dimensional space). Given two vectors $a$ and $b$, the inner product (dot product) of $a$ and $b$ is defined

$$
\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i b_i
$$

As you might expect, there are naïve and tail-recursive ways to compute this.

**List Inner Product: Tail recursive**

$$
\text{inner}(A,B,N) :- \text{dotaux}(A,B,0,N)
$$

$$
\text{dotaux}([], [], V, V).
\text{dotaux}([A|As],[B|Bs],N,Z) :-
N1 is N + (A * B),
\text{dotaux}(As, Bs, N1, Z).
$$

**List Inner Product : Naive**

$$
\text{inner}([], [], 0).
\text{inner}([A|As],[B|Bs],N) :-
\text{inner}(As,Bs,Ns),
N is Ns+(A * B).
$$

**Mutual recursion**

$$
\text{even}([]).
\text{even}([B|C]) :- \text{odd}(C).
\text{odd}([A|C]) :- \text{even}(C).
$$

Peter Norvig: Any mutual recursion can be converted to direct recursion using procedural inlining. *... some time it find difficult...*

$$
\text{even}([]).
\text{even}([A,B|C]) :- \text{even}(C).
\text{odd}([A]).
\text{odd}([A|C]) :- \text{odd}(C).
$$

**Mapping: The Full Map**

Map each list element (a number) to a term $s(A,B)$ where $A$ is the number and $B$ is its square.

Input: [1, 9, 15]
Output: $s(1, 1), s(9, 81), s(15, 225)$

$$
\text{sqterm}([], []).
\text{sqterm}([X|T], [s(X,Y)|L]) :-
Y is X * X,
\text{sqterm}(T, L).
$$
/* fullmap(In, Out) */

fullmap([], []).
fullmap([X|T], [Y|L]) :-
    transform(X, Y),
    fullmap(T, L).

/* Here is a typical transformation table...*/

transform(cat, gatto).
transform(dog, cane).
transform(hamster, criceto).
transform(X, X).

?- fullmap([cat, dog, goat], Z).
Z = [gatto, cane, goat]

Sometimes the map needs to be sensitive to the input data:

Input: [1, 3, w, 5, goat]
Output: [1, 9, w*w, 25, goat*goat]

Using the infix binary compound term *, it is easy enough to give some mathematical reality to the map:

Clause for Integer data

Input: [1, 3, w, 5, goat]
Output: [1, 9, w*w, 25, goat*goat]

Squared data Clause

Given an input list, partially map it to an output list.

evens([], []).
evens([X|T], [X|L]) :-
    0 is X mod 2,
    evens(T, L)
evens([X|T], L) :-
    1 is X mod 2,
    evens(T, L).

?- evens([1, 2, 3, 4, 5, 6], Q).
Q = [2, 4, 6].
Backtracking and Non-determinism

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

?- member(fred, [john, fred, paul, fred]).
   Yes

?- member(X, [john, fred, paul, fred]).
   X = john;          \textcolor{red}{\textbf{Deterministic query}}
   X = fred;
   X = paul;
   X = fred
   no

The problem of controlling backtracking

colour(cherry, red).
colour(banana, yellow).
colour(apple, red).
colour(apple, green).
colour(orange, orange).
colour(X, unknown).

?- colour(banana, X).
   X = yellow

?- colour(physalis, X).
   X = unknown

?- colour(cherry, X).
   X = red; \textcolor{red}{\textbf{(pressed semicolon)}}
   X = unknown;
   no

Generic one
True for every one
If there any other solution?

The cut: in Prolog

- A built-in predicate, used not for its logical properties, but for its effect. Gives control over backtracking.
- The cut is spelled:  \texttt{!}
- The cut always succeeds.
- When backtracking over a cut, the goal that caused the current procedure to be used fails.

How it works

- Suppose that goal \texttt{H} has two clauses:
  \texttt{H}\textsubscript{1} : \texttt{B}\textsubscript{1}, \texttt{B}\textsubscript{2}, ..., \texttt{B}\textsubscript{i}, !, \texttt{B}\textsubscript{k}, ..., \texttt{B}\textsubscript{m}.
  \texttt{H}\textsubscript{2} : \texttt{B}\textsubscript{n}, ... \texttt{B}\textsubscript{p}.

  - If \texttt{H}\textsubscript{1} matches, goals \texttt{B}\textsubscript{1}...\texttt{B}\textsubscript{i} may backtrack among themselves.
  - If \texttt{B}\textsubscript{i} fails, \texttt{H}\textsubscript{2} will be attempted. But as soon as the 'cut' is crossed, Prolog commits to the current choice. All other choices are discarded.

Commitment to the clause

Goals \texttt{B}\textsubscript{1}...\texttt{B}\textsubscript{m} may backtrack amongst themselves, but if goal \texttt{B}\textsubscript{i} fails, then the predicate fails (the subsequent clauses are not matched).

\texttt{H}\textsubscript{1} : \texttt{B}\textsubscript{1}, \texttt{B}\textsubscript{2}, ..., \texttt{B}\textsubscript{i}, !, \texttt{B}\textsubscript{k}, ..., \texttt{B}\textsubscript{m}.
\texttt{H}\textsubscript{2} : \texttt{B}\textsubscript{n}, ... \texttt{B}\textsubscript{p}.

How to remember it

Think of the 'cut' as a 'fence' which, when crossed as a success, asserts a commitment to the current solution.
However, when a failure tries to cross the fence, the entire goal is failed.
**Cut : Use 1**

To define a deterministic (functional) predicate.

A deterministic version of member, which is more efficient for doing ‘member checking’ because it needn’t give multiple solutions:

```prolog
membercheck(X, [X|_]) :- !.
membercheck(X, [_|L]) :- membercheck(X, L).
```

?- membercheck(fred, [joe, john, fred, paul]).
yes.

?- membercheck(X, [a, b, c]).
X = a; no.

**Max with cut**

max(X, Y, X) :- X >= Y, !.
max(X, Y, Y).

• If max is called with X >= Y, the first clause will succeed, and
  the cut assures that the second clause is never made.
• The advantage is that the test does not have to be made twice if X<Y.

The disadvantage is that each rule does not stand on its own as a logically correct statement about the predicate. To see why this is unwise, try

?- max(10, 0, 0).

**Cut : Use 2**

To specify exclusion of cases by ‘committing’ to the current choice.

The goal max(X, Y, Z) instantiates Z to the greater of X and Y:

max(X, Y, X) :- X >= Y.
max(X, Y, Y) :- X < Y.

Note that each clause is a logically correct statement about maxima. A version using cut can get rid of the second test, and might look like this:

max(X, Y, X) :- X >= Y, !.
max(X, Y, Y).

**Max with cut**

So, it is better to using the cut and both tests will give a program that backtracks correctly as well as trims unnecessary choices.

max(X, Y, X) :- X >= Y, !.
max(X, Y, Y) :- X < Y.

Or, if your clause order might suddenly change (because of automatic program rewriting), you might need:

max(X, Y, X) :- X >= Y, !.
max(X, Y, Y) :- X < Y, !.

**Cut: One more example**

rem_dups, without cut:

```prolog
rem_dups([], []).
rem_dups([F|Rest], NRest) :- member(F, Rest),
                     rem_dups(Rest, NRest).
rem_dups([F|Rest], [F|NRest]) :-
                      not(member(F, Rest)),
                     rem_dups(Rest, NRest).
```

rem_dups([], []).  
rem_dups([F|Rest], NRest) :- member(F, Rest),!
                      rem_dups(Rest, NRest).
rem_dups([F|Rest], [F|NRest]) :-
                      rem_dups(Rest, NewRest).

**Cut: One more example**

rem_dups, with cut:

```prolog
rem_dups([], []).
rem_dups([F|Rest], NRest) :-
                      member(F, Rest),!
                      rem_dups(Rest, NRest).
rem_dups([F|Rest], [F|NRest]) :-
                      not(member(F, Rest)),
                     rem_dups(Rest, NRest).
```
Thanks