More examples :

- John likes anyone who likes wine.

```
likes (john, X)  :-  likes (X, wine).
```

- John likes any female who likes wine.

```
likes (john, X) :- female (X), likes (X, wine).
```

- Defining sisterof (X, Y) predicate

```
sisterof (X, Y) :- female (X), parents (X, M, F),
                   parents (Y, M, F).
```

V Database

```
male (albert).
male (edward).
female (alice).
female (victoria).
parents (edward, victoria, albert).
parents (alice, victoria, albert).
sisterof (X, Y) :- female (X), parents (X, M, F),
```

goal: sisterof (alice, edward)

1. X b.t. alice
   Y b.t. edward

2. PROLOG tries to satisfy female (alice).

   female (alice) succeeds.

3. Tries to satisfy parents (alice, M, F)

   M b.t. victoria
   F b.t. albert

4. Tries to satisfy parents (edward, victoria, albert)

   Entire goal succeeds

   Yes

Next goal: Find out if alice is anyone's sister.

?- sisterof (alice, X).

1. X in the goal and Y in the clause are shared.

   X in the clause is bound to alice.

2. PROLOG tries to satisfy: female (alice) which succeeds

3. PROLOG tries to satisfy parents (alice, M, F) which succeeds with M b.t. victoria and F b.t. albert.

4. PROLOG tries to satisfy parents (Y, victoria, albert).

   It succeed with Y b.t. edward.

   So, a solution is "edward".

   If ";" is pressed (In Std. PROLOG), Then ?

Second solution generated is "alice" !!

To eliminate "alice" from the solution, we can write

sisterof (X, Y) :- female (X), parents (X, M, F), parents (Y, M, F), X <> Y.

- However, X <> Y can't be put immediately after female (X) as that would result into an error "Free variable in the expression".

Or alternatively in the goal section:

sisterof (alice, X), Y = alice, X <> Y.

X = edward, Y = alice

1 solution

We can't write sisterof (alice, X), X <> alice as these are objects of different types, as per PROLOG

Example

predicates

likes (symbol, symbol)

clauses

likes (marry, food).
likes (marry, flowers).
likes (john, food).
likes (john, food).
likes (john, wine).

I goal : likes (marry, X), likes (john, X).

X = food

X = food

2 solutions

II goal : likes (john, X), likes (marry, X).

X = food

X = food

2 solutions

III goal : likes (john, X), likes (john, X).

X = food

X = food

X = food

X = food

X = wine

5 solutions.

IV goal: likes(john, X), likes(marry, Y).

X = food Y = food

X = food Y = flowers

X = food Y = food

X = food Y = flowers

X = wine Y = food

X = wine Y = flowers

6 solutions.

Database

```
female (marry).

mother (john, ann).

mother (marry, ann).

father (marry, fred).

father (john, fred).

sisterof (X, Y) :- female (X), parent (X, M, F),
                   parent (Y, M, F).

parent (X, M, F) :- mother (X, M),
                    father (X, F).
```

goal: sisterof (marry, Y).

Y = john
Y = marry (Immediately after Y=marry, the 2nd solution is found, PROLOG returns to prompt. Further possibilities are not there, so not tried)

Question: How to exclude "marry" from the soln?

goal: sisterof (marry, Y), Y <> marry.

Error: Turbo-PROLOG says:
"Expression may not contain objects

goal: sisterof (marry, Y), x = marry, x <> Y

Y = john , x = marry 1 soln.

Example /* only for illustration. */

female (marry).

parent (john, ann, fred).

parent (john, lilly, fred).

parent (marry, ann, fred).

parent (marry, ann, bill).

sisterof (X, Y) :- female (X), parent (X, M, F),
parent (Y, M, F).

goal: sisterof (marry, X).

M = ann
F = fred
- X = john
- X = marry

M = ann
F = bill
→ X = marry

3 solutions

List

- List is an ordered sequence of elements, where order of elements matter.
- Elements of a list may be constant, variable or list themselves.
- List can contain an unlimited number of elements.
- All the objects in a list must be of the same type, but may be very complex.
- In LISP, the only data structure available is List apart from constants.
- The list can be bifurcated as Head and Tail. The notation is [Head | Tail].

Head – Contains the first element of a list

Tail – It's a list which contains all the elements except the Head.

Example

List	Head	Tail
[a, b, c]	a	[b, c]
[the, [cat, sat]]	the	[[cat, sat]]
[]	None	None

Matching of Lists

List 1	List 2	Instantiation
[X, Y, Z]	[john, likes, fish]	X = john Y = likes Z = fish
[cat]	[X | Y]	X = cat Y = []
[X, Y | Z]	[marry, likes, wine]	X = marry Y = likes Z = [wine]
[[the, Y] | Z]	[[X, hare], [is, here]]	X = the Y = hare Z = [[is, here]]
[vale, horse]	[horse, X]	Don't match

Declaring list in PROLOG

```
domains

    list = symbol*

predicates

    member (symbol, list)

clauses

    member (X, [X|_]).

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

goal: member (susan, [tom, bill, susan])
```

Call : member (susan, [tom, bill, susan])
Fail : member (susan, [tom, bill, susan])
Redo : member (susan, [tom, bill, susan])

- Call : member (susan, [bill, susan])
- Fail : member (susan, [bill, susan])
- Redo : member (susan, [bill, susan])
  - Call: member (susan, [susan])
  - Ret: member (susan, [susan])
- Ret : member (susan, [bill, susan])

Ret : member (susan, [tom, bill, susan])

True

Goal: writelist([a, b, c]).

a

b

c

Yes

```
domains
        list = symbol *

predicates
        writelist (list)

clauses
        writelist ([Head | Tail]) :- write (Head), nl,
                                     writelist (Tail).
```

Output:

a
b
c
No

How to eliminate "No" from the answer?

Soln: By writing writelist[] as last clause.

. Even, if we write writelist([]) over the given clause, still answer doesn't change.

In Recursive Definitions, we must look for:

(a) Left Recursion

This arises when a rule causes the invocation of a goal that is essentially equivalent to the original goal that caused the rule to be used.

Thus, if we defined:

```
mother (lilly, adam).

person (X) :- person (Y), mother (X, Y).

person (adam).
```

Goal: person (Z).

Problem ?

Backtracking is not possible, because in order to backtrack a clause must fail.

Soln: If we put person (adam) over a rule, then PROLOG has chance to look for finding solutions.

Heuristics: Put facts over rules in DB.

(b) Circular Definitions

```
parent (X, Y) :- child (Y, X).

child (X, Y) :- parent (Y, X).
```

Example 2

```
domains
        list = integer *

predicates
        testlist (list)

clauses
/* c1 */    testlist ([_ | Tail]) :- testlist(Tail).

/* c2 */    testlist ([]).
```

```
/*  goal: testlist ([_ | Tail]) :- testlist(Tail). */
```

~~Yes~~

```
goal:  testlist ([1,2]).
        Yes
goal:  testlist (X).
        Stack overflow
```

If we interchange c1 and c2, then o/p will be

X = []

Appending a list

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

Goal: append([a,b,c], [d,e,f], X)

X = [a,b,c,d,e,f]

1 solution.

Goal: append(X, [a,b,c], [d,e,a,b,c])

X = [d,e]

1 solution.

Execution Trace:

Call Goal: append(["a", "b"], ["c", "d"], ~~X~~)

Redo : append(["a", "b"], ["c", "d"], —)

Call: append(["b"], ["c", "d"], —)

Redo: append(["b"], ["c", "d"], —)

Call: append([], ["c", "d"], —)

Ret: append([], ["c", "d"], ["c", "d"

Ret: append(["b"], ["c", "d"], ["b", "c",

Ret : append(["a", "b"], ["c", "d"], ["a",
"b", "c", "d"])

Reversing a List

```
domains
        list = integer*

predicates
        reverse (list, list, list)

clauses
        newreverse ([], Inputlist, Inputlist).
        newreverse ([Head | Tail], List1, List2) :-
                newreverse (Tail, [Head | List1], List2).
```

goal: reverse ([1,2], [], Z)

Z = [2,1]

| solution

or if the goal is

```
reverse(List1, Z)
  :- newreverse(List1, [], Z).
```

{ goal: reverse (Z, [], [1,2]) /* Error */ }

Call: reverse ([1,2], [], —Z)

Redo: reverse ([1,2], [], —Z)

Call: reverse ([2], [1], — List2)

Redo: reverse ([2], [1], —)

Call: reverse ([], [2,1], —)

Ret: reverse ([], [2,1], [2,1])

Ret: reverse ([2], [1], [2,1])

Ret: reverse ([1,2], [], [2,1])

Rotate ~~Right~~ Left

```
rl([H|T], X) :- append(T, [H], X).
```

```
domains
        list = integer *
predicates
        last-element (list, integer)

clauses
        last_element ([Z], X) :- Z=X.
```

This can be written as last-element ([X], X).

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

Goal: last-element ([1,2,3], X)

X = 3

1 solution

Goal: last-element ([1,2,3], 3)

Yes

Goal: last-element ([], X)

No solution

What if we change 1st predicate : last_element (Z, Z).

With this change, if the goal is last-element ([1,2,3], X)

X = [1,2,3]

X = [2,3]

X = [3]

X = []

4 sol^{n}/

[Here, the declaration of the list should be last-element (list, list)]

Execution Trace of the last element with above change:

Call : l- element ([1,2,3], —)

Ret : l- element ([1,2,3], [1,2,3])
↑ First Soln ← First Clause

Redo : l-element ([1,2,3], —) ← Second Clause Call

Call : l- element ([2,3], —)

Ret : l- element ([2,3], [2,3]) ← First Clause succes

Ret : l- element ([1,2,3], [2,3])
↑ 2nd Soln ← Second Clause success

Redo : l- element ([2,3], —) ← Second clause

Call : l- element ([3], —)

Ret : l- element ([3], [3]) ← First Clause

Ret : l- element ([2,3], [3])

Ret : l-element ([1,2,3], [3])
↑ 3rd Solution

Redo : l-element ([3], —)

Call : l-element ([], —)

ret : l-element ([], []) ← First Clause

Ret : l- element ([3], [])

Ret : l- element ([2,3], [])

Ret : l- element ([1,2,3], [])
↑ 4th Soln

Redo : l- element ([], —) ← Second Clause

fail : l- element ([], —)

4 Solutions

Another Version of L-E

Last element using append

last-element (L, R) :-
append (—, [R], L).