NOVA IMS Information Management School

AI

Artificial Intelligence

Knowledge Representation and Reasoning

Prolog – part II

Marco Antonio Silva
madasi@microsoft.com

Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

An introduction to Prolog

- **Lists**

Lists are structures that represent sets of elements. Sets are enclosed in square brackets and elements separated by commas. Lists may be empty and may also contain sublists.

List Examples:

- empty list: [ ]
- List with elements: [1, 2, 4, 7, 8, 20, 3,10]
- List of lists: [[11, 111], [10, 16], [1, 2, 3]]

List operations are almost always based on a relationship between the list and its first element.

The separation of the first element of a list is achieved by writing: **[X | R]** . R can be anything, including an empty list.

For example, in the list [1, 3, 5 | R], X will be 1 and R will be [3, 5]

An introduction to Prolog

List Concept

A list can be empty or a sequence of elements or lists of lists. The rule that defines us a list is:

```
list([ ]).
list([X | Xs]):- list(Xs).
```

- list([1, 3, 9, [1, 2], [ ], [[1, 2], [7], 4]]).
  list([1, 3, 9, [1, 2], [ ], [[1, 2], [7], 4]])
  list([3, 9, [1, 2], [ ], [[1, 2], [7], 4]])
  list([9, [1, 2], [ ], [[1, 2], [7], 4]])
  list([[1, 2], [ ], [[1, 2], [7], 4]])
  list([[ ], [[1, 2], [7], 4]])
  list([[[1, 2], [7], 4]])
  list([ ])
  yes

An introduction to Prolog

- Check if an element belongs to a list

To check if an element belongs to a list, two predicates are constructed. The first works as a stop condition and the second runs through the list. An element belongs to a list if it is at the head of the list (stop condition).

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

If it is not in the head, it may be one of the other elements, so it becomes necessary to remove the head and check again if the next one (which will now be at the head of the list)

Removing the heads of the list is provisional, as, when the stop condition is checked or the list gets empty it is returned with the same elements as it was executed.

An introduction to Prolog

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

Example:

```
?- member(3,[2,3,4,5]).
   member(3,[2,3,4,5])
   member(3,[3,4,5])
   yes
```

An introduction to Prolog

- Prefixes and suffixes of a list

<u>One list is prefixed to another list when it is contained at the beginning of the second.</u>

The stop condition requires the first list to be empty. Until this happens, the "elimination" proceeds element by element, provided that they are equal.

```
prefix([ ], Tail).
prefix([X|Xs], [X | Ys]):- prefix(Xs, Ys)
```

An introduction to Prolog

```
prefix([ ], Tail).
prefix([X|Xs], [X | Ys]):- prefix(Xs, Ys)
```

Example:

```
?       prefix([1, 2, 3], [1, 2, 3, 4, 5]).
        prefix([1, 2, 3], [1, 2, 3, 4, 5])
        prefix([2, 3], [2, 3, 4, 5])
        prefix([3], [3, 4, 5]).
        prefix([ ], [4, 5])
        yes
```

An introduction to Prolog

One list is suffixed by another when its endings are the same.

To construct such a rule, it is necessary to remove elements from the second until they are equal, a situation represented by the stopping condition.

```
suffix(Xs, Xs).
suffix(Xs, [Y I Ys]):- suffix(Xs, Ys).
```

Example:

```
? suffix([1, 2, 3], [4, 1, 2, 3]).
   suffix([1, 2, 3], [4, 1, 2, 3])
   suffix([1,2, 3], [1, 2, 3])
   yes
```

Sublist of a list

This rule lets you demonstrate that list Xs is a sublist of Ys. The rule definition is made using the two rules previously defined: prefix and suffix.

Algorithm: Xs is a sublist of Ys if we get a prefix of Ys where Xs is suffix (suffix rule), or if we get a prefix of Xs that is suffix of Ys (prefix rule)).

```
sublist(Xs, Ys):- prefix(Zs, Ys), suffix(Xs, Zs).
sublist(Xs, Ys):- prefix(Xs, Ks), suffix(Ks, Ys).
```

An introduction to Prolog

```
sublist(Xs, Ys):- prefix(Zs, Ys), suffix(Xs, Zs).
sublist(Xs, Ys):- prefix(Xs, Ks), suffix(Ks, Ys).
```

Example:

```
?       sublist([2, 3], [1, 2, 3, 4]).
        sublist([2, 3], [1, 2, 3, 4])
        prefix([], [1, 2, 3, 4])
        suffix([2, 3], []),                    False.
        prefix([], [1, 2, 3, 4])
        prefix([1], [1, 2, 3, 4])
        prefix([1,2], [1, 2, 3, 4])
        prefix([1,2,3], [1, 2, 3, 4])
        yes
```

Cardinal from a list

Calculating the number of elements in a list can be implemented by eliminating element by element until the list is empty. At this point we know that the cardinal is zero and so we can initialize the counter. The total number of elements is achieved by take advantaged of back-tracking.

```
card([ ] , 0).
card([X | Xs], M):- card(Xs, N), M is N + 1.
```

An introduction to Prolog

```
card([ ] , 0).
card([X | Xs], M):- card(Xs, N), M is N + 1.
```

Example:

```
?- card([3, 4, 5],  X).
        card([3, 4, 5], X)
        card([4, 5],_)
        card([5],_)
        card([ ], 0)                Counter = zero
        1 is 0 + 1                  Back-Tracking.
        card([5], 1)
        2 is 1 + 1
        card([4, 5], 2)
        3 is 2 + 1
        card([3, 4, 5], 3)
        yes
```

An introduction to Prolog

Concatenate two lists

The concatenation of two lists gives rise to a third, which will be the result. This technique starts by removing the elements one by one from the first list until it is empty, that is: until we reach the stopping condition, which will unify the second list with the result list. This is where the elements from the second rule are added to the result list, giving rise to the final result..

```
conc([ ], Ys, Ys).
conc([X | Xs], Ys, [X | Zs]):- conc(Xs, Ys, Zs).
```

Example:

```
?       conc([1, 2], [3, 4], Ys).
        conc([1, 2], [3, 4], Ys)
        conc([2], [3, 4],_)
        conc([ ], [3, 4],_)
        conc([ ] ,[3, 4], [3, 4])
        conc([2], [3, 4], [2, 3, 4])
        conc([1, 2], [3, 4], [1, 2, 3, 4])
        yes
```

An introduction to Prolog

Invert a list

To invert a list, another predicate is invoked with an already initialized and empty list. The second rule is to remove element by element from the list to invert, placing it at the head of the list that will result by applying the stop condition, only written at the end. This rule unifies the second argument with the third.

```
invert(Xs, Ys) :- invert(Xs, [ ], Ys).
invert([X | Xs], Temp, Ys):- invert(Xs, [X | Temp], Ys).
invert([ ], Ys, Ys).
```

?	invert([1, 2, 3], Y).	
	invert([1, 2, 3], Y)	(by the 1st rule)
	invert([1, 2, 3], [ ], Y)	(by the 1st rule)
	invert([2, 3], [1], Y)	(by the 2nd rule)
	invert([3], [2, 1], Y)	(by the 2nd rule)
	invert([ ], [3, 2, 1], Y)	(by the 2nd rule)
	invert([ ],[3, 2, 1], [3, 2, 1])	(by the 3rd rule)
	….	
	….	
	invert([1, 2, 3], [3, 2, 1])	(by the 1st rule)
	yes	

An introduction to Prolog

Remove an element from a list

Removal of the element only occurs once, that is: if there are repeated elements, only the first one to be found is removed. This is shown by the stop condition that tells us that if the desired element is equal to the first in the list, then the result is the rest of that list. The second rule goes by removing element by element from our list and placing it at the head of the result list.

```
del([X | Xs], X, Xs).
del([Y | Ys], X, [Y | Zs]):- del(Ys, X, Zs).
```

Example:

```
? del([1, 2, 1, 3], 2, Y).
  del([1, 2, 1, 3], 2, Y)
  del([2, 1, 3], 2, _ )          (by the 1st rule)
  del([2,1,3],2,[1,3])           (by the 2nd rule)
  del([1, 2, 1, 3], 2, [1, 1, 3]) (by the 1st rule)
  yes
```

Remove all occurrences of an element

While in the previous case only the first element to be found was removed from the list, In this case it is shown how to proceed in case the intended deletion is total. This requires one more rule in our previous set as well as a slight modification of them. The algorithm starts by removing all elements from the list until it is empty. At this point, the unification - stopping condition - is made which initializes the empty result list. During back-tracking, only elements other than the intended removal are added to the result list.

```
deltot([X | Xs], X, Ys):-  deltot(Xs, X, Ys).
deltot([X | Xs], Z, [X | Ys]):-  x <> Z,
deltot(Xs,Z, Ys).
deltot([ ], X, [ ]).
```

An introduction to Prolog

```
deltot([X | Xs], X, Ys):-  deltot(Xs, X, Ys).

deltot([X | Xs], Z, [X | Ys]):-  x <> Z,   deltot(Xs,Z, Ys).

deltot([ ], X, [ ]).
```

Example:

```
? deltot([1, 2,1, 3], 1, Ys).
   deltot([1, 2,1, 3], 1, Ys)          (by the 1st rule)
   deltot([2,1, 3], 1, Ys)             (by the 2nd rule)
   2 <> 1                              (by the 2nd rule)
   deltot([1, 3], 1, _)                (by the 1st rule)
   deltot([3], 1, _)                   (by the 2nd rule)
   3 <> 1                              (by the 2nd rule)
   deltot([ ], 1, Y)                   (by the 3rd rule)
  deltot([ ], 1, [ ])                  (by the 3rd rule)
  deltot([3],1, [3])
  deltot([1, 3],1, [3])
  deltot([2,1, 3],1, [2,3])
  deltot([l, 2,1, 3],1, [2,3])
  yes
```

An introduction to Prolog

Get the first element of a list

It is a very elementary predicate. It is possible, however, to elaborate a complex rule than the one presented.

```
first(X, [X | Xs]).
```

The first element of the list is what is at the head of the list.

Example:

```
first(F,[1,2,3]).
first(F, [1, 2, 3])
first(1, [1, 2,3])
yes
```

An introduction to Prolog

Get the last element of a list

This rule is also very simple. It is contrary to the previous one and its construction is summarized in the application of the invert predicate..

```
last(X, Xs):- invert(Xs, [Xr]), first(X,Xr).
```

Simplifying:

```
last(X, Xs):- invert(Xs, [X | Ys]).
```

An introduction to Prolog

Determine if a list is sorted

A list with an element is always sorted. This assumption serves as the stopping condition in the construction of this new rule. All that remains is to add an item to an ordered list or require that a particular element in the list be greater than or equal to the previous one. The change becomes immediate to remove two elements at a time, comparing them and re-invoking a rule with the second element inserted in the head of our list.

```
ordered([X]).
ordered([X, Y Ys])
X =< Y,
ordered([Y | Ys]).
```

Example:

```
ordered([1, 2, 3]).
ordered([1, 2, 3])
1 =< 2
ordered([2, 3])
2 =< 3
ordered([3])
ordered([1, 2, 3])
yes
```

An introduction to Prolog

Inserting an element in order

Inserting an element into an empty list results in a third list. It will also serve as a stopping condition as well as the third rule. The second rule aims to remove the elements from the list, as long as the removed element is smaller than the one to be added. The third acts as a stop rule and also adds the element to the list if it is greater than the element being tested in the list. This rule only runs if there are elements in the list larger than intended for addition..

```
insert(X, [ ], X).
insert(X, [Y | Ys], [Y | ZS])
X > Y,
insert(X, YS, ZS).
insert(X, [Y | Ys], [x, Y | Ys])
X =< Y,
```

Example:

```
?       insert(2,[1, 3], Y),
        insert(2,[1, 3], Y)
        2 > 1
        insert(2,[3],_)
        2 > 3
        2 =< 3
        Insert(2, [3], [2, 3])
        Insert[(2, [1, 3], [1, 2, 3])
        yes
```

An introduction to Prolog

Determine the largest element of a list

A cumbersome method of finding the largest element in a list is its sorting and subsequent determination of the last element. Although little or nothing efficient due to the need to sort and reverse the list, other less elegant solutions can be found.

The following is a more effective method based on the elaboration of two predicates which, although they have the same name, are different because the number of arguments varies.

NOTE: It is essential to inspect all elements of our list.

```
max([X | Xs], M):- max(Xs, X, M).
max([X | Xsl, Y, M):- X > Y, max(Xs, X, M).
max([X | Xsl, Y, M):- X =< Y,  max(Xs, Y, M).
max([ ], M, M).
```

An introduction to Prolog

```
max([X | Xs], M):- max(Xs, X, M).
max([X | Xsl, Y, M):- X > Y, max(Xs, X, M).
max([X | Xsl, Y, M):- X =< Y,  max(Xs, Y, M).
max([ ], M, M).
```

Example:

```
?- max([1, 3, 2] , M).
   max([1, 3, 2],  M)
   max([3, 2], 1,  M)
   3 > 1
   max([2], 3, M)
   2 > 3
   2 =< 3
   max([],  3, M)
   max([], 3, 3)
   max([2], 3, 3)
   max([3, 2], 1, 3)
   max([1, 3, 2], 3)
   yes
```

Determine the smallest element of a list

This search technique is analogous to the previous one only by changing the sign of the element comparison. It is also developed with the assumption that the first element in the list is the smallest.

```
min([X | Xs], M) :- min(Xs, X, M).
min([X |Xs], Y, M):- X =< Y, min(Xs, X, M).
min([X | Xs], Y, M):- X > Y, min(Xs, Y, M).
min([ ], M, M).
```

An introduction to Prolog

Replacing all occurrences of one element with another

In this particular case, the substitution will be made by changing element E by element N.

The technique of this predicate is achieved by removing all elements from our empty list when the result list is initialized. The second rule substitutes the element and the third is invoked whenever the second fails, ie when there are no substitutions to make.

```
subs([], E, N, []).

subs([E | Xs], E, N, [N | Ys]):- subs(Xs, E, N, Ys).

subs([X | Xs], E, N, [X |Ys])

subs(Xs, E, N, Ys).
```

The displayed debug shows replacing all “1” in the list with “3”

```
?       subs([1, 2, 1], 1, 3, Y).
        subs([1, 2, 1], 1, 3, Y)
        subs([ 2, 1], 1, 3,_)
        subs([1], 1, 3, _)
        subs([], 1, 3,_ )
        subs([], 1, 3, [])
        subs([1], 1, 3, [3])
        subs([2, 1], 1, 3, [2, 3])
        subs([1, 2, 1], 1, 3, [3, 2, 3])
        yes
```

An introduction to Prolog

Union of two lists by removing repeated elements

Behind referred or predicate conc which concatenated two lists. The predicate, in addition to concatenating, also eliminates repeated elements, both those appearing in the same list and those in both lists.

```
union([ ], Ys, Ys).

union([X | Xs], Ys, Zs):- not(member(X, Ys)), union(Xs, [X | Ys], Zs).

union([X | Xs], Ys, Zs):- union(Xs, Ys, Zs).
```

Debug:

```
?-  union([1, 2, 2, 3], [1, 2, 4], Y).
    union([1, 2, 2, 3], [1, 2, 4],  Y)

    member(1, [1, 2, 4]),
    union([2, 2, 3], [1, 2, 4], Y)
    member(2, [1, 2, 4])
    union([2, 3], [1, 2, 4], Y)
    member(2, [1, 2, 4])
    union([3], [1, 2, 4], Y)
    member(3, [1, 2, 4])
    union([], [3, 1, 2, 4], Y)
    union([], [3, 1, 2, 41, [3, 1, 2, 4])
    union([3],    [1, 2, 4], [3, 1, 2, 4])
    union([2, 3], [1, 2, 4], [3, 1, 2, 4])
    union([2, 2, 3], [1, 2, 4], [3, 1, 2, 4])
    union([1, 2, 2, 3], [1, 2, 4], [3, 1, 2, 4])
```

An introduction to Prolog

Operator **..**

Converts a structure to a list, or a list to a structure.

Sintax: **Struct = .. List:**

Example:

```
? - a(b, c, d) = .. B.
    B = [a, b, c, d]
    yes
```

or on the contrary:

```
? - A = ..[a, b, c].
    A = a(b, c)
    yes
```

Calculation of the greatest common divisorr between two digits.

```
read (A)
read (B)
while (A<>B)
if A > B then A ← A - B
if A < B then B ← B - A
end while
write (A)
```

Euclid's algorithm

Greatest Common Divisor (gcd) of two or more non-zero integers is the largest positive integer that divides the numbers without a remainder.

An introduction to Prolog

```
main/0.
mcn/0,
read-number/0.

ifthen(X,(Y,Z)):-X,Y,Z.

read-number/0.

main:- write('***********great common number divisor - 2 numbers ***********'),
      nl,
      write('number--> '),
      read(N),
      nl,
      write('number2 -->'),
      read(M),
      nl,
      gcd(N, M, X).


gcd(0, X, X) :- write(X),!.
gcd(X, 0, X) :- write(X),!.
gcd(X, X, X) :- write(X),!.
gcd(M, N, X) :- N>M, Y is N-M, gcd(M, Y, X).
gcd(M, N, X) :- N<M, Y is M-N, gcd(Y, N, X).
```

main/0 - if the files that are loaded from the command line contain a main/0 predicate, then it is called immediately.

An introduction to Prolog

```
main/0.
mcn/0,
read-number/0.

ifthen(X,(Y,Z)):-X,Y,Z.
```

main/0 - if the files that are loaded from the command line contain a main/0 predicate, then it is called immediately.

```
main:-
writewrite('**********great common number divisor - n numbers **********'),
nl,
write(" Enter a number sequence ending in 0 "),
nl,
read_number,
mcn.

read_number:-
  write("number: "),
  read(N),
  ifthen(N<>0,
    (assert(number(N)),
     read_number
  ).
```

An introduction to Prolog

```
mcn :-
number(X),
number(Y),
X > Y,
NewX is X - Y,
retract(number(X)),
assert(number(NewX)),
mcn.
```

```
mcn:-
number(X),
write("Maximum Common Number divisor"),
write(X).
```

An introduction to Prolog

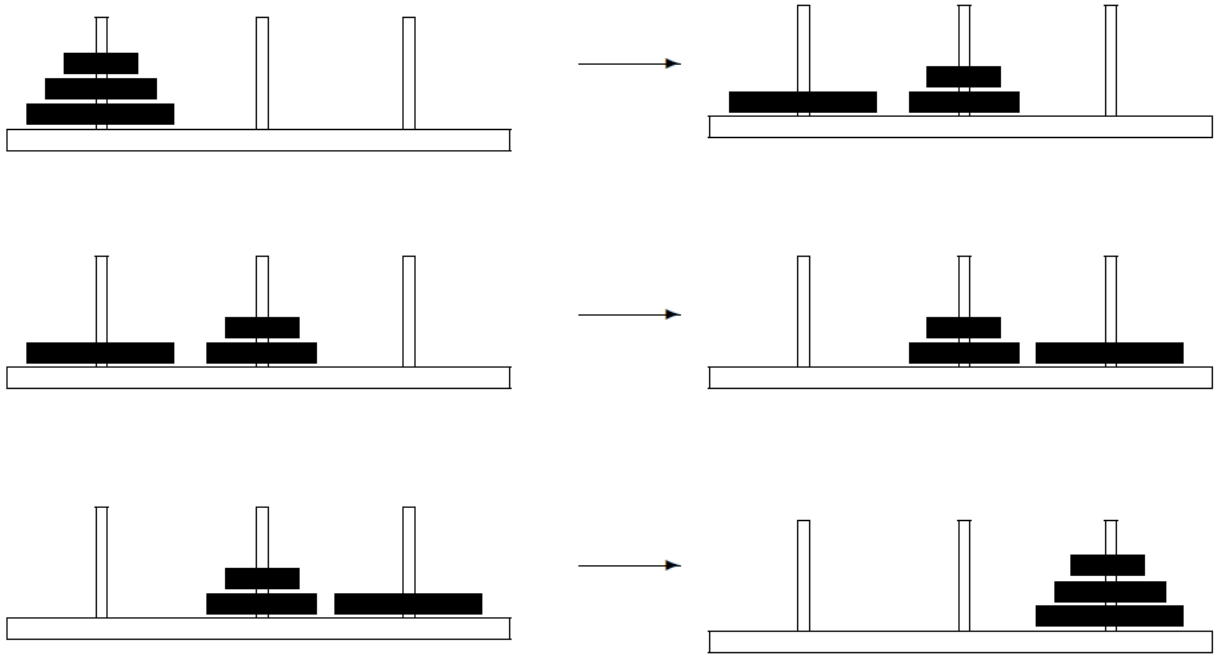
Examples of programs in Prolog

Calculation of the distance between arches

```
arc(a,b,3).
arc(b,d,2).
arc(b,e,5).
arc(e,c,2).
arc(e,f,6).
arc(f,d,1).

path(X,Y) :- arc(X,Y).
path(X,Y) :- arc(X,Z), path(Z,Y).

path(X,Y,[Y]) :- arc(X,Y).
path(X,Y,[Z|T]) :- arc(X,Z), path(Z,Y,T).

path(X,Y,[Y],D) :- arc(X,Y,D).
path(X,Y,[Z|T],V) :- arcX,Z,M), path(Z,Y,T,C),V is M+C.
```

Vitor Santos

Example – Hanoi´s Towers



Example – Hanoi´s Towers

```
hanoi :-
   write("How many discs?"),
   nl,
   read(N),
   number(N),
   begin(N),
   move(N, e, d, c).

begin(N) :-
   write("Hanoi Tower Solution for'),
   write(N),
   write(' Discs: '),
    nl.
```

```
move(1, origin, destiny, _) :-
   write(origin),
   write(' -> '),
   write(destiny),
    nl.

move(N, origin, destiny, aux) :-
    N > 1,
    M is N-1,
   move(M, origin, aux, destiny),
   move(1, origin, destiny, _),
   move(M, aux, destiny, origin).
```

Example - Three House Problem

- There is a street with three houses, all in different colors, one house is blue, one is red and another is green. In each house lives a person with a different nationality than people living in the other houses.
- In each house there is a pet and the pets are different in every house.
- The Englishman lives in the red house. Spanish's pet is a budgie.
- The Japanese lives in the house to the right of the person who has a fish. The person who has a fish lives to the left of the blue house.
- Who has a turtle?

Example - Three House Problem

- Before solving this prologue puzzle, we began to display the information involved. For this, we will define two structures, street and house definition (*house*), as follows:

```
house (C, street (C,,)).
house (C, street (, C,)).
house (C, street (,, C)).
```

Example - Three House Problem

We also define a predicate position of three arguments. The literal position (R, C1, C2) states that on R street, house C1 is to the left of house C2, or, what is the same, that house C2 is to the right of house C1. If E and D are houses, we have:

position (street (E, D,), E, D).
position (street (, E, D), E, D).

The default structure is a term composed of three arguments.

The expression *house* (C, N, A) represents a house of color C, inhabited by the person of nationality N and who has pet A.

We now have to define three predicates of two arguments color, nationality and animal whose meanings are translated by the following statements:

colour(house(C, ,), C).
nationality (house(, N,), N).
animal(house(, , A), A).

Example - Three House Problem

- A representation of the clues provided by the puzzle:
- The Englishman lives in the Red House::
  - house (I, Street), color (I, red), nationality (I, English).
- The Spanish's pet is a budgie:
  - house(E, Street), animal(E, budgie), nationality(E, Spanish).
- The Japanese live in the house that is on the right side of the person who has a fish:
  - house(J, Street), house(P1, Street), nationality(J, Japanese), animal(P1, fish), position(Street, P1, J).
- Person who has fish lives in the house that is on the left of blue house:
  - house(P1, Street), house(P2, Street), animal(P1, peixe), color(P2, blue), position(Street, P1, P2).

Example - Three House Problem

The following prolog program provides a solution to this puzzle. :

```
solve :- clues(Street),
question(Street).


clues(Street) :-
house(I, Street),
color(I, red),
nationality(I, english),
house(E, Street),
animal(E, budgie),
nationality(E, spanish),
house(J, Street),
house(P1, Street),
nationality(J, japanese),
animal(P1, fish),
position(Street, P1, J),
house(P2, Street),
color(P2, blue),
position(Street, P1, P2).
```

```
question(Street) :-
house(X, Street),
animal(X, turtle),
nationality(X, Nat),
write('The'),
write(Nat),
write(' have the turtle.'),
nl.


color(defhouse(C, _, _), C).
nationality(house(_, N, _), N).
animal(house(_, _, A), A).
position(Street(E, D, _), E, D).
position(Street(_, E, D), E, D).
house(X, Street(X, _, _)).
house(X, Street(_, X, _)).
house(X, Street(_, _, X)).
```

With this program we get the interaction:
? - solves.
The Japanese have the turtle.

Example - DIAGNOSTIC SYSTEM

```
start:-
        cls,
        write('      **************************************'),nl,
        write('      **************************************'),nl,
        write('      *          * DIAGNOSTIC SYSTEM*       * '),nl,
        write('      *              * CAR FAILURES*        * '),nl,
        write('      **************************************'),nl,
        write('      **************************************'),nl,
        write('      *    * ANSWERS MUST BE YES or NO. *     *'),nl,
        write('      ** DON'T FORGET THE "." IN THE END OF ANSWERS**'),nl,
        write('          ** THEN PRESS ENTER **'),nl,
        write('      **************************************'),nl,
        write('      **************************************'),nl,
        nl,nl,nl,!, cfailure.
```

Example - DIAGNOSTIC SYSTEM

```
eng_sta(X):-
        write(The engine starts?'),nl,
        write('Answer->'),
        read(Answer),asserta(eng_sta(Answer)),nl,!,X=Answer.

prob_bat(X):-
        write('Insufficient speed of the starter?'),nl,
        write('Answer->'),
        read(Answer),asserta(prob_bat(Answer)),nl,!,X=Answer.

prob_spark(X):-
        write('Spark plugs disabled?'),nl,
        write('Answer->'),
        read(Answer),asserta(prob_spark(Answer)),nl,!,X= Answer.
```

Example - DIAGNOSTIC SYSTEM

```
cfailure :-
        car_ failure(State),nl,
        write('          " DIAGNOSIS"   '),
        maybe(State).

/******** Does not start  **********/
car_ failure(batery):-                  /*1*/
                        eng_sta(no),
                        prob_bat(yes).


car_ failure(spark _def):-              /*2*/
                        eng_sta(no),
                        prob_bat(no),
                        prob_spark(yes).
```

```
car_ failure(voltage_ spark):-          /*3*/
                        eng_sta(no),
                        prob_bat(no),
                        prob_spark(no),
                        high_voltage(yes).

car_ failure(filter):-                  /*4*/
                        eng_sta(no),
                        prob_bat(no),
                        prob_spark(no),
                        high_voltage(no),
                        prob_filter(yes).
```

Practical work Example – Programming in Prolog

- Mobile phone troubleshooting
- Exercise ->
- Build a mini diagnostic system for mobile failures. Implement a simple user interface
- Discussion

Bibliography

- Leon Sterling and Ehud Shapiro, The Art of Prolog
- Mathematical Logic for Computer Science , M. Ben-Ari, 2001, Springer-Verlag
- Logic in Computer Science: Modelling and Reasoning about Systems , M. Huth e M. Ryan, 2004, Cambridge University Press
- Visual Prolog Reference Guide and resources:

  http://wiki.visual-prolog.com/

  http://www.visual-prolog.com/