# Artificial Intelligence

# Lab sheet No:4

## Backtracking

As has already been seen prolog has built in backtracking mechanism. It tries to prove a goal with all possible instantiations. Automatic backtracking is a useful programming concept because it reveals the programmer of the burden of backtracking explicitly. However in some cases this feature degrades the efficiency of the program.

For example in cases where one solution is sufficient, backtracking to find all the solutions is not a good idea. Similarly, in case of mutually exclusive rules(clauses) when one rule has been proved then it is known in advance that no other rules can succeed. So this backtracking can be controlled by the use of ‘cut’, (“!”).

The disadvantage of using cut is that we tend to move away from the declarative nature of the prolog because when we have used the cut the order of the clauses may make difference in the result we get.

There are two main uses of the cut:

1. When you know in advance that certain possibilities will never give rise to meaningful solutions, it's a waste of time and storage space to look for alternate solutions. If you use a cut in this situation, your resulting program will run quicker and use less memory. This is called a green cut.
2. When the logic of a program demands the cut, to prevent consideration of alternate subgoals. This is a red cut

Let’s consider an examples that show how you can use the cut in your programs.

**Example Rule : r1 :- a, b, !, c.**

This is a way of telling Visual Prolog that you are satisfied with the first solution it finds to the subgoals a and b. Although Visual Prolog is able to find multiple solutions to the call to c through backtracking, it is not allowed to backtrack across the cut to find an alternate solution to the calls a or b. It is also not allowed to backtrack to another clause that defines the predicate r1.

**Example Program**

```
PREDICATES
buy_car(symbol,symbol)
nondeterm car(symbol,symbol,integer)
colors(symbol,symbol)
CLAUSES
buy_car(Model,Color):-
car(Model,Color,Price),
colors(Color,sexy),!,
Price < 25000.
car(maserati,green,25000).
car(corvette,black,24000).
car(corvette,red,26000).
car(porsche,red,24000).
colors(red,sexy).
colors(black,mean).
colors(green,preppy).
GOAL
buy_car(corvette, Y).
```

In this example, the goal is to find a Corvette with a sexy color and a price that's ostensibly affordable. The cut in the buy_car rule means that, since there is only one Corvette with a sexy color in the known facts, if its price is too high there's no need to search for another car.

Given the goal

*buy_car(corvette, Y)*

1. Visual Prolog calls car, the first subgoal to the buy_car predicate.

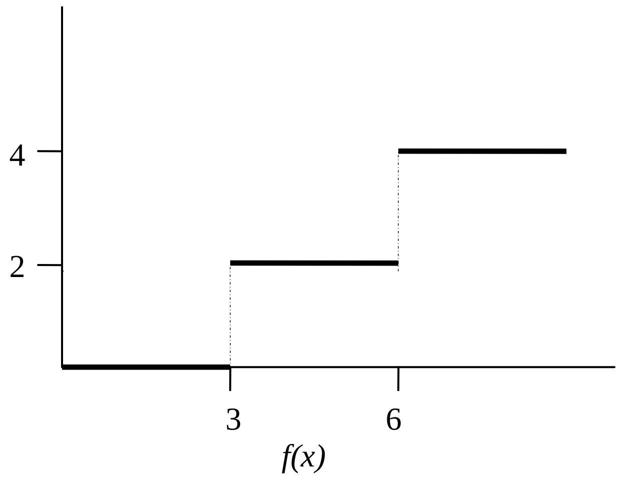
2. It makes a test on the first car, the Maserati, which fails.
3. It then tests the next car clauses and finds a match, binding the variable Color with the value black.
4. It proceeds to the next call and tests to see whether the car chosen has a sexy color. Black is not a sexy color in the program, so the test fails.
5. Visual Prolog backtracks to the call to car and once again looks for a Corvette to meet the criteria.
6. It finds a match and again tests the color. This time the color is sexy, and Visual Prolog proceeds to the next subgoal in the rule: the cut. The cut immediately succeeds and effectively "freezes into place" the variable bindings previously made in this clause.
7. Visual Prolog now proceeds to the next (and final) subgoal in the rule: the comparison *Price < 25000.*
8. This test fails, and Visual Prolog attempts to backtrack in order to find another car to test. Since the cut prevents backtracking, there is no other way to solve the final subgoal, and the goal terminates in failure

Consider the function as shown in the figure below. The relation between X and Y can be specified by the following three rules.

Rule 1: if X<3 then Y=0

Rule 2: if 3=<X <6 then Y=2

Rule 3: if 6<X then Y=4



A double step function

This can be programmed as

```
PREDICATES
f(integer,integer)

CLAUSES
f(X,0):- X<3.
f(X,2):- 3<=X,X<6.
f(X,4):- 6<X.

GOAL
f(2,X).
```

Assignment 1.) Now modify the program using cut and observe the difference between the two modules. Comment on the difference.

Assignment 2.) Define the relation **min(X,Y,Z)** where Z returns the smaller of the two given numbers X and Y. Do it with and without the use of cut and comment on the result.

## Structure revisited

In prolog we can use structures to define data types of our requirement. For example if we want to use date as an structure we can define date as a structure in the domains section as follows

date=d(integer,symbol,integer)

We can then on use date as a data type to contain the date.

```
DOMAINS
date=d(integer,symbol,integer)

PREDICATES
inquire
display(symbol)
date_of_birth(symbol,date)
```

```
CLAUSES
date_of_birth(ram,d(12,july,1983)).
date_of_birth(shyam,d(15,august,1976)).
date_of_birth(hari,d(26,may,1994)).
date_of_birth(sita,d(29,september,1991)).

display(X):-
        date_of_birth(X,Y),
        write(X),nl,
        write(Y).
inquire:-
        write("Enter the name"),
        readln(X),
        display(X).
GOAL
inquire.
```

Here the goal so proceeds as to ask a name from the user and to display the date of birth of the person with that name. With a little modification we can write goals which can find out persons with age below or above certain value, persons born in a month etc as in a relational database.

So the facts of the prolog can be thought of as a database. In fact we use structures to define certain relations and for all purposes of integrity this can be used similar to a table in a relational database. We call it the prolog's internal database. We can update this database during the execution of the program by using the following keywords.

**assert(C) – this keyword can be used to assert a data in the facts base as**
**asserta(C) and assertz( C) can be used to control the position of insertion, the two asserts at the beginning and the end respectively.**
**retract( C) –deletes a clause that matches C.**

**An example**

```
DOMAINS
date=d(integer,symbol,integer)
works=w(symbol,integer)

FACTS
person(symbol,symbol,date,works).

PREDICATES
start
load_name
evalans(integer)
display
search
dispname(symbol)
delete

CLAUSES
person(shyam,sharma,d(12,august,1976),w(ntv,18000)
).
person(ram,sharma,d(12,august,1976),w(ntv,18000)).
person(ram,singh,d(13,may,2001),w(utl,12000)).

start:-
        write("************MENU**********
***"),nl,
        write("Press 1 to add new data"),nl,
        write("Press 2 to show existing data"),nl,
        write("Press 3 to search"),nl,
        write("Press 4 to delete"),nl,
        write("Press 0 to exit"),nl,
        write("************MENU**********
***"),nl,
        readint(X),
        evalans(X).

evalans(1):-load_name,start.
evalans(2):-display,evalans(2).
evalans(3):-search,evalans(3).
evalans(4):-delete,evalans(4).
evalans(0):-write("Thank You").

delete./* write clauses delete a fact from the facts base
*/
search./*write    clauses to search a fact from the
facts base */

dispname(N):-
        person(N,C,d(D,M,Y),w(O,S)),
        write("Name:",N," ",C),nl,
        write("Date of Birth:",D,"th"," ",M," ",Y),nl,
        write("Organisation:",O),nl,
        write("Salary:",S),nl,nl.

display:-
        retract(person(N,X,d(D,M,Y),w(O,S))),
        write("Name:",N," ",X),nl,
        write("Date of Birth:",D,"th"," ",M," ",Y),nl,
        write("Organisation:",O),nl,
        write("Salary:",S),nl,nl.

load_name:-
        write("Enter the name \n"),
        readln(N),
        write("Enter the surname \n"),
        readln(S),
        write("Date of Birth \n Day:"),
        readint(D),nl,
        write("Month:"),
        readln(M),nl,
        write("Year:"),
        readint(Y),nl,
        write("Enter the organisation:"),
        readln(O),
        write("Enter the salary:"),
        readint(Sl),nl,nl,
        asserta(person(N,S,d(D,M,Y),w(O,Sl))).

GOAL
start.
```

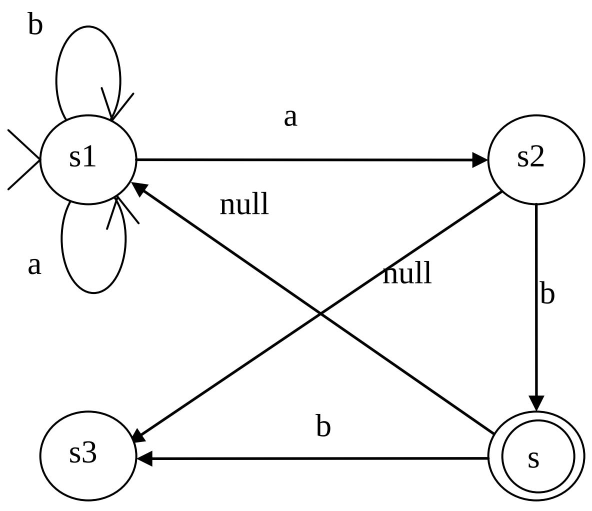
We may not use prolog to handle databases but the use of prologs internal database makes problem solving with prolog lot more easy.

Assignment: Observe the above program. Add clauses for search and delete and extend your module as much as you like 3.

## Simple application

Let us now see how the features of prolog can be used to simulate a non-deterministic automata which would have been a cumbersome task using other programming languages.

A nondeterministic finite automaton is an abstract machine that reads a string of symbols as input and decides whether to accept or to reject the input string. An automaton has a number of states and it is always in one of the states. The automata can change from one state to another upon encountering some input symbols. In a non deterministic automata the transitions can be non deterministic meaning that the transition may take place with a NULL character or the same character may result in different sitions
A non deterministic automaton decides which of the possible moves to execute, and it chooses a move that leads to the acceptance of the string if such a move is available.
Let us simulate the given automata.



```
DOMAINS
Symb_list=symbol*

PREDICATES
Trans(symbol,symbol,symbol)
Silent(symbol,symbol)
Final(symbol)

CLAUSES
final(s3).

trans(s1,a,s1).
trans(s1,a,s2).
trans(s1,b,s1).
trans(s2,b,s3).
trans(s3,b,s4).
```

```
silent(s2,s4).
silent(s3,s1).

accepts(S,[]):- final(S).

accepts(S,[H|T]):-
        trans(S,H,S1),
        accepts(S1,T).

accepts(S,X):-
        silent(S,S1),
        accepts(S1,X).

GOAL
Accepts(S,[a,b]).
```

Assignment 4.) Check the automaton with various input strings and with various initial states. ( The initial state need not necessarily be s1.) Observe the result and comment on how the simulation works.

Use the following goals

- accepts(s1,[a,a,b]).
- accepts(s1,[a,b,b]).
- accepts(S,[b,a,b]).
- accepts(s1,[X,Y,Z]).
- accepts(s2,[b]).
- accepts(s1,[_,_,_,_|[a,b]]).