Q1:a

Best first search algorithm:

```
begin
   open := [Start];                                                    % initialize
   closed := [ ];
   while open ≠ [ ] do                                                 % states remain
      begin
         remove the leftmost state from open, call it X;
         if X = goal then return the path from Start to X
         else begin
                generate children of X;
                for each child of X do
                case
                     the child is not on open or closed:
                        begin
                             assign the child a heuristic value;
                             add the child to open
                        end;
                     the child is already on open:
                        if the child was reached by a shorter path
                        then give the state on open the shorter path
                     the child is already on closed:
                        if the child was reached by a shorter path then
                          begin
                             remove the state from closed;
                             add the child to open
                          end;
                end;                                                   % case
                put X on closed;
                re-order states on open by heuristic merit (best leftmost)
             end;
return FAIL                                                            % open is empty
end.
```

1. Open=[A10]; closed=[]
2. Evaluate A10; open=[C20,B30,D40];closed=[A10]
3. Evaluate C20; open=[B30,G35,D40,H40]; closed=[C20, A10]
4. Evaluate B30; open=[E10,F20,G35,D40,H40]; closed=[B30,C20,A10]
5. Evaluate E10; open=[L10,F20,G35,K40,D40,H40];closed=[E10,B30,C20,A10]
6. Evaluate L10; open=[Q20,F20, G35,K40,D40,H40];closed=[L10,E10,B30,C20,A10]
7. Evaluate Q20; the solution is found.

Q1#b

Hill climbing search algorithm:

1. Generate a possible solution, see if it is a solution, if so quit else continue.
2. From this solution apply some number of applicable rules to generate a new set of proposed solutions(states).
3. For each element of the set do the following:
   3.1.Send it to a test function, if t is a solution then quit.

3.2. If not, see if it is closest to the solution of any of the elements tested so far, if it is remember it, else forget it.

4. Take the best element(state) found above and use it as the next proposed solution.
5. Go back to step2.

The path by using hill climbing algorithm is: ACGMQ

Q2#

a) Count the number of words that contain "tion" in a given list.

```
Domains
slist=string*
Predicates
 count(string)
 set(slist,integer)
clauses
count(S):-frontstr(4,S,X,_),X="tion",!.
count(S):-frontchar(S,_,R),count(R).
set([],0).
set([H|T],L):-count(H),set(T,L1),L=L1+1,!.
set([_|T],L):-set(T,L).
goal
set(["ab","tion"],X).
```

trace

```
set(["ab","tion"],X).
count("ab")
frontstr(4,"ab",X,_) --- fail
frontchar("ab",_,R),
count("b")
frontstr(4,"b",X,_) --- fail
frontchar("b",_,R)
count("")
frontstr(4,"",X,_) ---- fail
frontchar("",_,X) ---- fail
count("ab") ---- fail
set(["tion"],L)
count("tion")
frontstr(4,"tion",X,_),
X="tion" ---- true
Count("tion") ---- true
Set([],L1) --- true
L1=0
L=0+1
L=1
Set(["ab","tion"],X) ---- true
```

X=1.

b) Count the number of characters in a given string using external database

```
database
single sum(integer)
predicates
count(string)
clauses
count(""):-sum(X),write(X),!.
count(S):-frontchar(S,_,S1),sum(X),X1=X+1,assert(sum(X1)),count(S1).
Goal
assert(sum(0)),
count("yes").
```

<u>trace</u>

```
count("yes")
 frontchar("yes",_,S1) --- true
S1="es"
sum(X)
X=0
X1=1
sssert(sum(1))
count("es")
frontchar("es",_,S1) --- true
S1="s"
sum(X)
X=1
X1=2
assert(sum(2))
count("s")
frontchar("s",_,S1) --- true
S1=""
sum(X)
X=2
X1=3
assert(sum(3))
count("")
sum(X)
X=3
write(3) ---- output
count("yes") --- true
```

c) Find how many times an element occurs in a list using non tail recursion.

```
domains
ilist=integer*
predicates
count(integer,ilist, integer)
clauses
count(_,[],0):-!.
count(H,[H|T],S):-count(H,T,S1), S=S1+1,!.
count(X,[_|T],S):-count(X,T,S).
Goal
count(3,[2,3,4,3],X).
```

trace

count(3,[2,3,4,3],X)

count(3,[3,4,3],S)

count(3,[4,3],S1)

count(3,[3],S)

count(3,[],S1)---- true

S1=0

S=S1+1

S=0+1

S=S1+1

S=1+1

count(3,[2,3,4,3],X) ---- true

X=2.

Q3#

Domains

ilist=integer*

name,city_name,street_no_house_no,class=string

age=i(integer);r(real)

```
address=add(city_name,street_no,house_no)

degrees=ilist

student=st(name,age,address,degrees,class)

students=student*

Database

data(student)

predicates

read_data(integer)

read_deg(ilist)

start

find_average

sum_count(ilist,integer,integer)

clauses

start:-read_data(20),find_average,!.

read_data(0):-!.

read_data(N):-write("enter the name"),readln(N),nl,write("enter the age"),
                readterm(age,A), nl,write("enter the address"),
                readterm(address,Add),nl,read_deg(L), nl,write("enter the
                class"), readln(C), asserta(data(st(N,A,Add,L,C))), N1=N-1,
                read_data(N1).

read_deg([H|T]):-write("enter the degree"), readint(H), H>=0, nl, read_deg(T).

read_deg([]):-!.

find_average:-data(stu(N,_,_,D,_)), sum_count (D,S,N),Av=S/N,
                write("the average of ",N),write("is  ",Av),fail.

find_average:-!.

sum_count([],0,0):-!.
sum_count([H|T],S,N):- sum_count(T,S1,N1),S=S1+H, N=N1+1,!.
```

Q4#

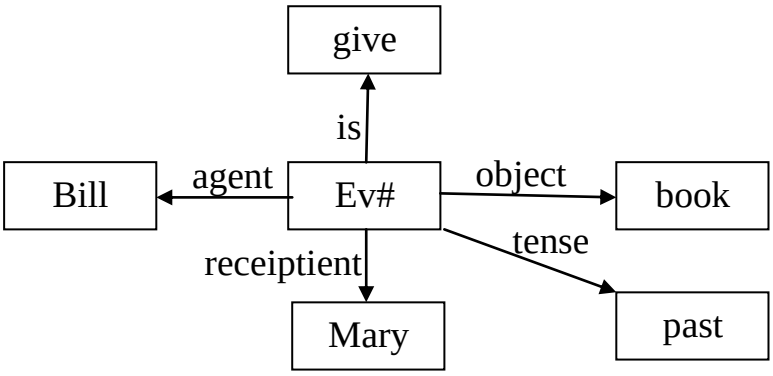
In Al, there are four basic categories of representational schemes: logical, procedural, network and structured representation schemes.

1. **Logical representation** uses expressions in formal logic to represent its knowledge base. Predicate Calculus is the most widely used representation scheme, also there is propositional calculus. In prepositional calculus, each atomic symbol (P, Q, etc.) denotes a proposition of some complexity. There is no way to access the components of an individual assertion. Predicate calculus provides this ability. For example, instead of letting a single prepositional symbol, P, denote :the entire sentence "it rained on Tuesday," we can create a predicate weather that describes a relationship between a date and the weather. weather (Tuesday, rain) through inference rules we can manipulate predicate calculus expression accessing their individual components and inferring new sentences. Predicate calculus also allows expressions contain variables. Variables let us create general assertions about classes of entities. For example, we could state that for :all values, of X, where X is a day of the week , the statement weather (X, rain) is true ; I,e., it rains it rains every day.
2. **Procedural representation** represents knowledge as a set of instructions for solving a problem. These are usually if-then rules we use in rule-based systems.
   Production rules provide a formal way of representing recommendations, directives or strategies, they are often appropriate when the domains knowledge results from empirical association developed through years of experience solving problems in a particular field.

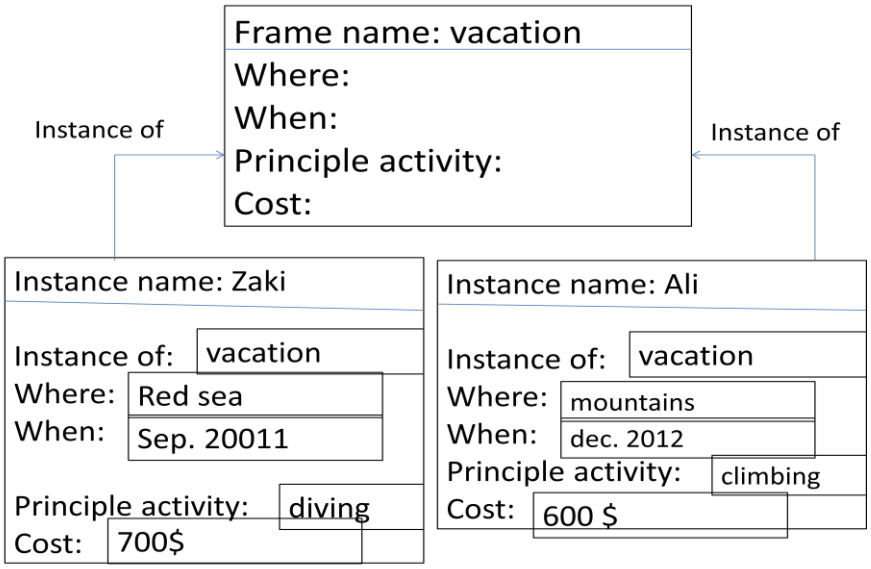
   Example: IF a flammable liquid was spilled THEN call the fire departments.

3. **Network representation** captures knowledge as a graph in which the nodes represent objects or concepts in the problem domain and the arcs represent relations or associations between them.
   The term semantic nets is used to describe a knowledge representation method based on a network structure. Semantic nets were originally developed for use as psychological models of human memory but are now a standard representation method for AI and expert systems.

   semantic net have been used successful in natural language understanding, for example, the sentence “Bill gave the book to Mary”



4. **Structured representation** extends network representation schemes by allowing each node to have complex data structures named slots with attached values.
   In order to expand the capabilities of semantic networks an object in semantic networks is expanded to be a frame, every frame has some properties, the frame representation follow the slot/filler approach. Inheritance could be applied to frames where frame inherit properties of another frame. Some frames may be permanent in the system others may be created and destroyed during the course of problem solving.
   Example:



<u>Q5#</u>

a) Predicate form and clause form.

The predicate form can be convert to clause form using the following steps:

1. Eliminate the → by using: $a \rightarrow b \equiv \neg a \vee b$.
2. Reduce the scope of negation.
3. Standardize by renaming all variables so that variables bound by different quantifiers have unique names.
4. Move all quantifiers to the left without changing their order.
5. All existential quantifiers are eliminated by a process called skolemization.
6. Drop all universal quantification.
7. Convert the expression to the conjunct of disjunctions form.
8. Call each conjunct a separate clause.

9. Standardize the variables apart again. This requires giving. The variable in each clause generated by step 8 different names.
   As an example if we have the sentence "All people who are not poor and are smart are happy", will be converted to predicate form as:
   $\forall x(\neg poor(X) \wedge smart(X) \rightarrow happy(X))$ and its clause form will be:
   Poor(X) V ¬smart(X)V happy(X)

b) A- algorithm and A* algorithm.

The first advance approach to the best first search is known as A-search algorithm. A algorithm is simply define as a best first search plus specific function. This specific function represent the actual distance (levels) between the initial state and the current state and is denoted by g(n). A notice will be mentioned here that the same steps that are used in the best first search are used in an A algorithm but in addition to the g(n) as follow: F(n) = h(n) + g(n) where h(n) is a heuristic function. The second advance approach to the best first search is known as A*-search algorithm. A* algorithm is simply define as a best first search plus specific function. This specific function represent the actual distance (levels) between the current state and the goal state and is denoted by h(n). A* search is the most widely-known form of best-first search. It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal: f(n) = g(n) + h(n).
Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have f (n) is equal to the estimated cost of the cheapest solution through n.

c) Backtracking and recursion.

Backtracking
Prolog tries to find all solution by using its ability in backtrack, in prolog we have two predicates that control the backtracking (cut and fail).
1. Cut
Cut is used in prolog to turn off backtracking, that represented by an exclamation point (!). The cut effectively tells Prolog to freeze all the decisions made so far in this predicate. That is, if required to backtrack, it will automatically fail without trying other alternatives.

<u>For example:</u>
```
a(1).
a(2).
a(4).
b(2).
b(3).
d(X,Y):-a(X),!,b(Y), X=4.
```
Goal: d(X,Y)

X=1, Y=2, 1=4 ----Fail
X=1, Y=3, 1=4 ----Fail
Output: no solution.

2. Fail

The fail predicate is provided by Prolog. When it is called, it causes the failure of the rule. And this will be forever; nothing can change the statement of this predicate.

For example:

```
clauses
a(1).
a(2).
a(3).
a(4).
Begin:-a(X),write(X),fail.
Goal: Begin
Output: 1234 NO
```

Recursion

The recursion in any language is a function that can call itself until the goal has been succeed. In Prolog, recursion appears when a predicate contain a goal that refers to itself. There are two types of recursion.

1. **Tail Recursion**: is recursion in which the recursive call is the last sub goal of the last clause. That is, in tail recursion the recursive call is always made just before the procedure exits: the last step. For example: tail_recursion(X):- b, c, tail_recursion(Y).
2. **Non Tail Recursion:** when a recursive call is not last we don't have tail recursion. If you think about it, tail recursive calls can be implemented very efficiently because they can reuse the current stack frame or activation record rather than pushing a new one, filling it with data, doing some computation, and popping it. So tail recursion is often a fast option. For example: nontail_recursion(X):- b, nontail_recursion(X), c.

d) Forward chaining and backward chaining.

A state space may be searched in two directions, from the given data toward the goal or from the goal back to the data.

In data driven (also called forward chaining), the problem solver begins with the given facts of the problem and a set of rules or legal moves for chaining a state. Search proceed by applying rules to facts to produce new facts, which are in turn used by the rules to generate more facts. This process continues until it generates a path that satisfies the goal condition (state).

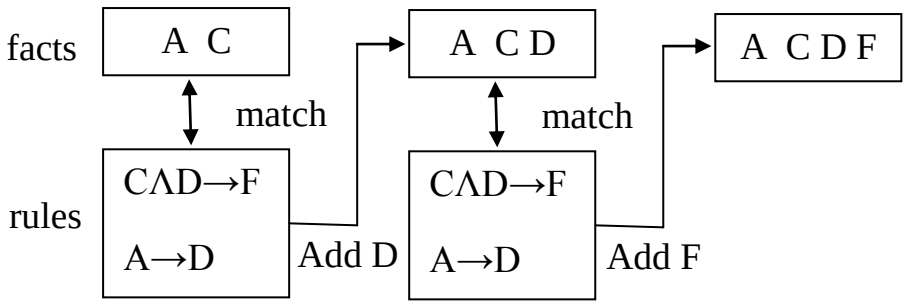
Goal driven search (also called backward chaining) focus on the goal, find the rules that could produce the goal and chain backward through successive rules and sub goals to the given facts of the problem.

Both data-driven and goal-driven problems solves the same state space graph, however the order and the actual number of states search can differ.

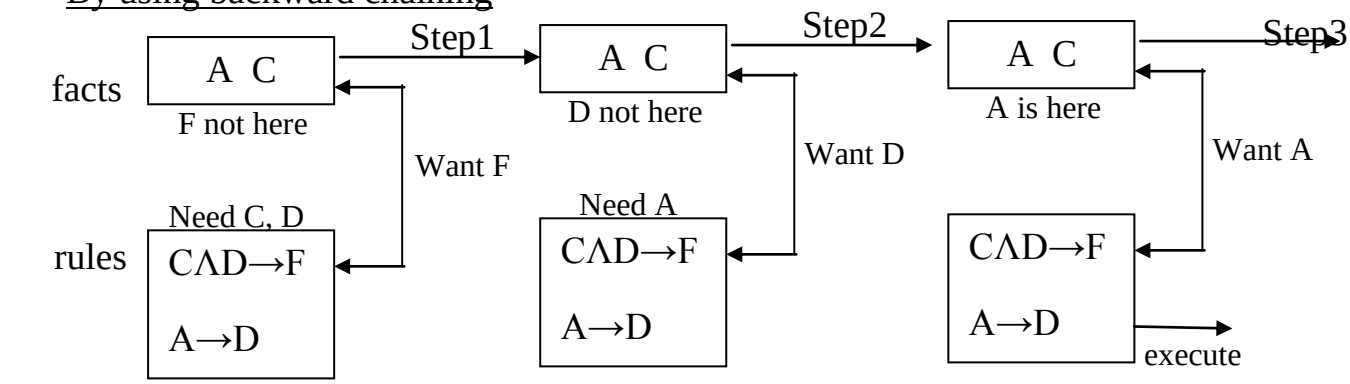
<u>For example</u>

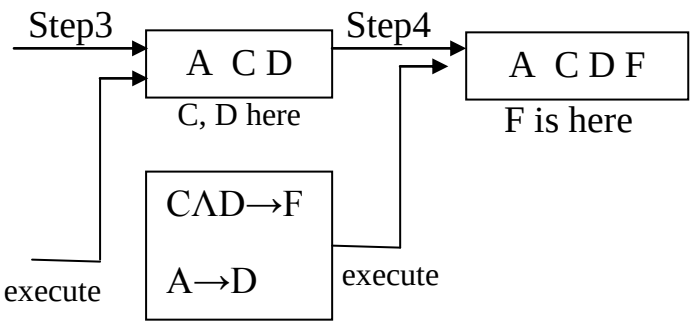
Given the facts A,C and rules A→D, CΛD→F prove F.

<u>By using forward chaining</u>



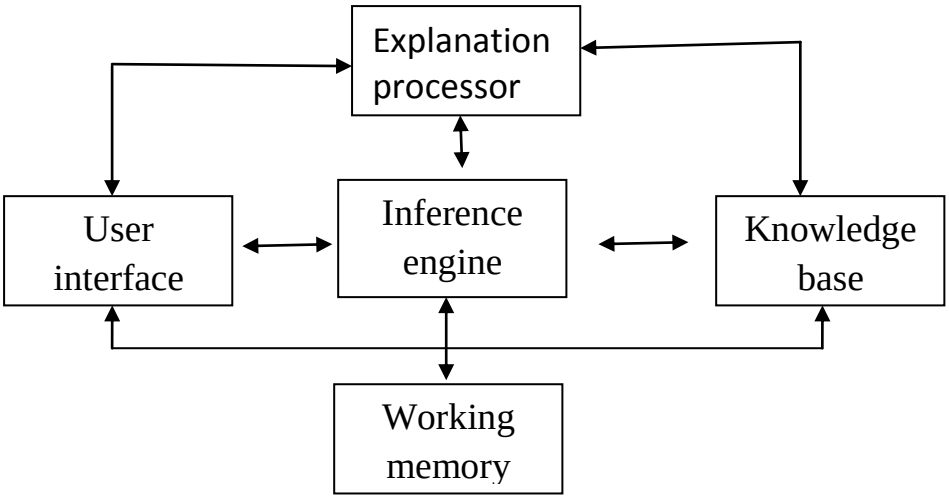
<u>By using backward chaining</u>





<u>Q6#</u>

Expert systems: are computer programs that are constructed to do the kinds of activities that human experts can do such as design, compose, plan, diagnose, interpret, summarize, audit, give advice.
The architecture of the expert system consists of several components as shown in figure below:



Expert system architecture

User Interface

The user interacts with the expert system through a user interface that make access more comfortable for the human and hides much of the system complexity. The interface styles includes questions and answers, menu-driver, natural languages, or graphics interfaces.

Explanation Processor

The explanation part allows the program to explain its reasoning to the user. These explanations include justifications for the system's conclusion (HOW queries), explanation of why the system needs a particular piece of data (WHY queries).

Knowledge Base

The heart of the expert system contains the problem solving knowledge (which defined as an original collection of processed information) of the particular applications, this knowledge is represented in several ways such as if-then rules form.

Inference Engine

The inference engine applies the knowledge to the solution of actual problems. It s the interpreter for the knowledge base. The inference engine performs the recognize act control cycle.

The inference engine consists of the following components:-

1. Rule interpreter.
2. Scheduler
3. HOW process
4. WHY process
5. knowledge base interface.

Working Memory

It is a part of memory used for matching rules and calculation. When the work is finished this memory will be raised.