

Program No.1

Turbo Prolog features and format.

Prolog is a logic programming language associated with artificial intelligence and computational linguistics. It has its roots in first order logic, a formal logic, and unlike many other programming languages. It is intended primarily as a declarative programming language, the program logic is expressed in terms of relations, represented as facts and rules.

The official version of Prolog was developed at the university of Marseilles, France by Alain Colmerauer in the early 1970s as convenient tool for programming in logic.

Turbo Prolog is a typed Prolog compiler, which means that it is much faster than the interpreted Prolog.

Difference between Turbo Prolog and other programming languages:

1. Name and structures of objects involved in the problem.
2. Name of relation which are known to exist between the objects.
3. Facts and rules describing these relations.

Execution of Turbo Prolog programs is controlled automatically. When a Turbo Prolog program is executed, the system tries to find all possible sets of values that satisfy the given goal. During execution , results may be displayed or the user may be prompted to type in some data. turbo prolog use a backtracking mechanism which, once one solution has been found ,causes Turbo Prolog to reevaluate any assumption made to see if some new variable values will provide new solutions.

Turbo Prolog has a very Short and simple syntax.

Turbo Prolog is powerful. Turbo Prolog typically uses ten times fewer program lines when solving a problem than language like Pascal.

Turbo Prolog has a built in pattern recognition facility, as well as simple and efficient way of handling recursive structures.

Turbo Prolog is compiled, yet allows interactive program development.



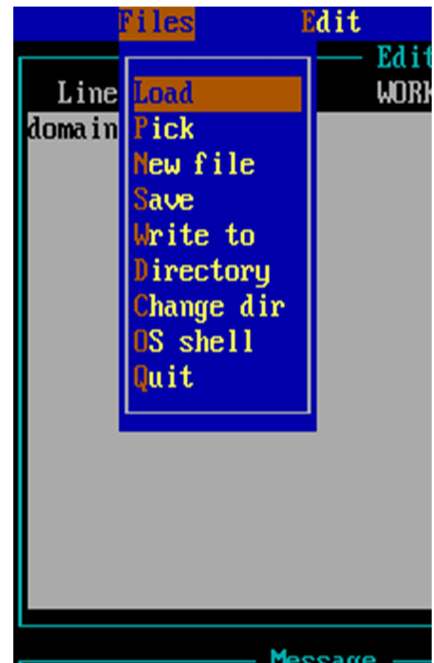
WINDOWS IN TURBO PROLOG

Turbo Prolog provides four windows-for views on the programming environment:-

- The editor window
It is the window where developer edits the text currently in the work file.
- The dialog window
It is the system window where the external goals are given or queries are made by the user and the results of those are recorded.
- The trace window
It is the window in which turbo Prolog can generate a trace of program execution.
The tracing the execution of a Prolog query allows you to see all of the goals that are executed as the part of the query, in sequence, along with whether or not they succeed. Tracing also allows you to see what steps occur as Prolog backtracks.
- The message window
It is the window in which messages related to the operation of the turbo prolog system appears.

Turbo Prolog Menu System:

- The Run Command
It is used to execute a program residing in memory.
- The Compile Command
It compiles the command currently in the editor.
- The Edit Command
It invokes the built-in editor for editing the file defined as the workfile.
- The File Menu
It has pull-down menus--
 - Load
It selects a workfile from the PRO directory.
 - Save
It saves the current workfile in the directory.
 - Directory
it is used to select the default directory for .PRO files. It can also be used to browse in the .PRO directory.
 - Quit
It is selected to leave the turbo Prolog system.
- Setup menu
it is selected when any of the setup parameters are to be inspected, temporarily changed or recorded permanently in a .SYSfile.



Parts of Turbo Prolog program

1. Compiler directives
2. Domains
3. Global domains
4. Database
5. Predicates
6. Global predicates
7. Goal
8. Clauses

A Prolog program consists of a number of clauses. each clause is either a fact or rule. After Prolog program is loaded in a Prolog interpreter, user can submit goals or queries, a the Prolog interpreter will give results according to the facts and rule.

Facts

A fact must start with a predicate and end with a full stop. The predicate may be followed by one or more arguments which are enclosed by parentheses. The arguments can be atoms, numbers, variables or lists. Arguments are separated by commas. In a Prolog program a presence of fact indicate that is true an absence of a fact indicates a statement that is not true.

For example:- father(kishan, vishal)

Rules

Rule can be viewed as an extension of fact with added conditions that also have to be satisfied for it to be true. It consists of two parts-the first part is similar to a fact and the second part consists of other clauses which must all be true for the rule itself to be true. These two parts are separated by ":-". you may interpret at this operator as "if" in English.

For example:-grandfather(X,Y):-father(X,Z),parent(Z,Y).

When a Prolog interpreter is running, you may probably see the prompt "?-" on the screen. This prompts the user to enter a goal or query.

Goals

Goal is statement starting with a predicate and probably followed by its arguments. The purpose of submitting a goal is to find out whether the statement represented by the goal is true according to the knowledge database.in valid goal, the predicate must have appeared in at least one fact or rule in the consulted program, and the number of arguments in the goal must be the same as that appears in the consulted program. Also, all the arguments are constant.

Queries

A query is a statement starting with a predicate and followed by its arguments, some of which are variables. Similar to goals, the predicate of a valid query must have appeared in at least one fact or rule in the consulted program, and the number of arguments in the query must be same as that appears in the consulted program. The purpose of submitting a query is to find values to substitute into the variables in the query such that the query is satisfied.

Turbo Prolog program example:-

DOMAINS

```
name = string
person,activity = symbol
```

PREDICATES

```
is(person,name)
likes(person,activity)
```

CLAUSES

```
likes(kishan,soccer).
likes(rahul,nda).
likes(bharat,cp).
likes(kanu,cricket).

likes(aditya,X) if likes(kanu,X).

likes(Y,gaming) if is(Y,"name").

is(bharat,"name").
is(kishan,"name").
```

Output:

```
Goal: likes(aditya,X)
X=cricket
1 Solution
Goal: likes(X,gaming).
X=bharat
X=kishan
2 Solutions
Goal: likes(kanu,nda).
No
Goal: likes(rahul,nda).
Yes
```

Program No.2

Write a program for usage of rules in Prolog.

DOMAINS

```
subject = symbol
teacher = symbol
code = integer
```

PREDICATES

```
teaches(teacher,subject)
subcode(subject,code)
tcode(teacher,code)
```

CLAUSES

```
subcode(os,9).
subcode(c,10).
subcode(webd,5).
subcode(cg,1).
tcode(ds,10).
tcode(ab,4).
tcode(pp,1).
teaches(X,Y) if subcode(Y,A),tcode(X,B),A<=B.
```

Output:

```
Goal: teaches(X,os).  
X=ds  
1 Solution  
Goal: teaches(pp,X).  
X=cg  
1 Solution  
Goal: teaches(ds,X).  
X=os  
X=c  
X=webd  
X=cg  
4 Solutions
```

Program No.3

Write a program for using Input, Output and fail predicates in Prolog.

PREDICATES

hello

GOAL

hello.

CLAUSES

hello:-

```
makewindow(1,7,7,"MY FIRST PROGRAM",4,56,10,22),  
nl,write(" ENTER YOUR NAME "),  
cursor(4,5),  
readln(Name),nl,  
write( "WELCOME " ,Name," !").
```


Output:

```
MY FIRST PROGRAM
ENTER YOUR NAME

ABHINANDAN
WELCOME ABHINANDAN !
Press the SPACE bar
```

Program No.4

Write a program for studying usage if arithmetic operators in Prolog.

PREDICATES

hello

GOAL

hello.

CLAUSES

hello:-

```
write(" Enter First Number "),
readreal(X),nl,
write(" Enter Second Number "),
readreal(Y),
Z = X + Y,
nl,write( " Sum is " ,Z),nl,
K = X - Y,
nl,write( " Difference is " ,K),nl,
T = X * Y,
nl,write( " Multiplication is " ,T),nl,
V = X / Y,
nl,write( " Division is " ,V).
```

Output:

```
Enter First Number 12
Enter Second Number 6
Sum is 18
Difference is 6
Multiplication is 72
Division is 2
```

Program No.5

Write a program to study usage of Cut, Not, Fail predicates in Prolog.

I. Using Cut predicate

DOMAINS

n,f = integer

PREDICATES

factorial(n,f)

go

GOAL

go.

CLAUSES

factorial(1,1) if !.

factorial(N,Res) if

N1 = N-1 and

factorial(N1,Between) and

Res = N*Between.

go:-

write("Enter the number "),

readreal(X),nl,

factorial(X,Y),

write("Factorial of ",X),

write(" is ",Y).

Output:

```
Enter the number 5  
Factorial of 5 is 120
```

II. Using Fail predicate

DOMAINS

name = symbol

PREDICATES

father(name,name)

everybody

GOAL

everybody.

CLAUSES

father(kartik,amar).

father(aman,rahul).

father(aman,mohan).

everybody if

father(X,Y) and

write(X," is ",Y,"'s father. \n") and fail.

Output:

```
kartik is amar's father.  
aman is rahul's father.  
aman is mohan's father.
```

III. Using Not predicate

DOMAINS

person = symbol

PREDICATES

male(person)

smoker(person)

vegetarian(person)

is_healthy(person)

GOAL

is_healthy(X) and

write("A possible Healthy Male is ",X),nl.

CLAUSES

male(bharat).

male(kishan).

male(rahul).

smoker(op).

smoker(rahul).

vegetarian(bharat).

vegetarian(rahul).

is_healthy(X) if male(X) and not(smoker(X)).

is_healthy(X) if male(X) and vegetarian(X).

Output:

A possible Healthy Male
is bharat

Program No.6

Write a program to study usage of recursion in Prolog.

DOMAINS

n,f = integer

PREDICATES

factorial(n,f)

go

GOAL

go.

CLAUSES

factorial(1,1).

factorial(N,Res) if

N > 0 and

N1 = N-1 and

factorial(N1,FacN1) and

Res = N*FacN1.

go:-

write("Enter the number"),nl,

readreal(X),

factorial(X,Y),

write("The factorial of ",X),

write(" is ",Y).

Output:

```
Enter the number 6
```

```
Factorial of 6 is 720
```

Program No.7

Write a program to implement DFS/BFS.

I. Breadth First Search

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 100
#define initial 1
#define waiting 2
#define visited 3

int n;
int adj[MAX][MAX];
int state[MAX];
void create_graph();
void BF_Traversal();
void BFS(int v);

int queue[MAX], front = -1, rear = -1;
void insert_queue(int vertex);
int delete_queue();
int isEmpty_queue();

int main()
{
    create_graph();
    BF_Traversal();
    return 0;
}

void BF_Traversal()
{
    int v;
    for(v=0; v<n; v++)
        state[v] = initial;
    printf("Enter Start Vertex for BFS: \n");
    scanf("%d", &v);
    BFS(v);
}

void BFS(int v)
{
    int i;
    insert_queue(v);
    state[v] = waiting;
    while(!isEmpty_queue())
```

```

    {
        v = delete_queue( );
        printf("%d ",v);
        state[v] = visited;
        for(i=0; i<n; i++)
        {
            if(adj[v][i] == 1 && state[i] == initial)
            {
                insert_queue(i);
                state[i] = waiting;
            }
        }
        printf("\n");
    }
}

```

```

void insert_queue(int vertex)
{
    if(rear == MAX-1)
        printf("Queue Overflow\n");
    else
    {
        if(front == -1)
            front = 0;
        rear = rear+1;
        queue[rear] = vertex ;
    }
}

```

```

int isEmpty_queue()
{
    if(front == -1 || front > rear)
        return 1;
    else
        return 0;
}

```

```

int delete_queue()
{
    int delete_item;
    if(front == -1 || front > rear)
    {
        printf("Queue Underflow\n");
        exit(1);
    }
    delete_item = queue[front];
}

```

```

        front = front+1;
        return delete_item;
    }

void create_graph()
{
    int count,max_edge,origin,destin;
    printf("Enter number of vertices : ");
    scanf("%d",&n);
    printf("Enter number of edges:");
    scanf("%d",&max_edge);
    printf("Note: GRAPH SHOULD BE CONNECTED\n");
    printf("The edge name starts from '0' and goes to 'n-1'\n");
    printf("For example\nEnter an edge(u,v):0 1    connects first and second node\nEnter an edge(u,v):0 n-1
    connects first and last node\n");

    for(count=1; count<=max_edge; count++)
    {
        printf("Enter an edge(u,v):");
        scanf("%d %d",&origin,&destin);
        if(origin>=n || destin>=n || origin<0 || destin<0)
        {
            printf("Invalid edge!\n");
            count--;
        }
        else
        {
            adj[origin][destin] = 1;
        }
    }
}

```

Output:

```
Enter number of vertices : 7
Enter number of edges: 6
Note: GRAPH SHOULD BE CONNECTED
The edge name starts from '0' and goes to 'n-1'
For example
Enter an edge(u,v):0 1          connects first and second node
Enter an edge(u,v):0 n-1       connects first and last node
Enter an edge(u,v):0 1
Enter an edge(u,v):0 2
Enter an edge(u,v):1 3
Enter an edge(u,v):1 5
Enter an edge(u,v):2 6
Enter an edge(u,v):2 4
Enter Start Vertex for BFS:
0
0 1 2 3 5 4 6
```

II. Depth First Search

```
#include<stdio.h>
#include<stdlib.h>

typedef struct node
{
    struct node *next;
    int vertex;
}node;

node *G[20];
//heads of linked list
int visited[20];
int n;
void read_graph();
//create adjacency list
void insert(int,int);
//insert an edge (vi,vj) in te adjacency list
void DFS(int);

void main()
{
    int i;
    read_graph();
    //initialised visited to 0
    for(i=0;i<n;i++)
        visited[i]=0;
    DFS(0);
}

void DFS(int i)
{
    node *p;
    printf("\n%d",i);
    p=G[i];
    visited[i]=1;
    while(p!=NULL)
    {
        i=p->vertex;
        if(!visited[i])
            DFS(i);
        p=p->next;
    }
}

void read_graph()
{
```



```

int i,vi,vj,no_of_edges;
printf("Enter number of vertices:");
scanf("%d",&n);
//initialise G[] with a null
for(i=0;i<n;i++)
{
    G[i]=NULL;
    //read edges and insert them in G[]
    printf("Enter number of edges:");
    scanf("%d",&no_of_edges);
    printf("Note: GRAPH SHOULD BE CONNECTED");
    printf("The edge name starts from '0' and goes to 'n-1'\n");
    printf("For example\nEnter an edge(u,v):0 1    connects first and second node\n
Enter an edge(u,v):0 n-1    connects first and last node\n");
    for(i=0;i<no_of_edges;i++)
    {
        printf("Enter an edge(u,v):");
        scanf("%d%d",&vi,&vj);
        insert(vi,vj);
    }
}
printf("\n");
}

```

```

void insert(int vi,int vj)
{
    node *p,*q;
    //acquire memory for the new node
    q=(node*)malloc(sizeof(node));
    q->vertex=vj;
    q->next=NULL;
    //insert the node in the linked list number vi
    if(G[vi]==NULL)
        G[vi]=q;
    else
    {
        //go to end of the linked list
        p=G[vi];
        while(p->next!=NULL)
            p=p->next;
        p->next=q;
    }
}
}

```

Output:

```
Enter number of vertices: 7
Enter number of edges: 6
Note: GRAPH SHOULD BE CONNECTEDThe edge name starts from '0' and goes to 'n-1'
For example
Enter an edge(u,v):0 1          connects first and second node
Enter an edge(u,v):0 n-1        connects first and last node
Enter an edge(u,v):0 1
Enter an edge(u,v):0 2
Enter an edge(u,v):1 3
Enter an edge(u,v):1 5
Enter an edge(u,v):2 6
Enter an edge(u,v):2 4

0
1
3
5
2
6
4
```

Program No.8

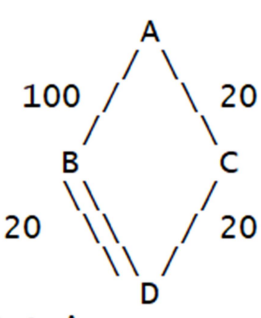
Write a program to implement A* algorithm.

```
from astar import AStar
class Path_finder(AStar):
    def __init__(self):
        self.nodes = {'A': [('B', 100), ('C', 20)], 'C': [('D', 20)], 'D': [('B', 20)], 'B': [('D',20)]}

    def neighbors(self,n):
        for n1, d in self.nodes[n]:
            yield n1
    def distance_between(self,n1, n2):
        for n, d in self.nodes[n1]:
            if n == n2:
                return d
    def heuristic_cost_estimate(self,n, goal):
        return 1
print("The graph is :")
print("          A")
print("         / \ ")
print("       100 /   \ 20")
print("         /     \ ")
print("        B       C ")
print("       \\      / ")
print("      20 \\    / 20 ")
print("         \\   / ")
print("          D")
start_node = input("Enter the start node for searching : ").upper()
end_node = input("Enter the end node for searching : ").upper()
Obj_Path_finder = Path_finder()
path = list(Obj_Path_finder.astar(start_node,end_node))
for i in path:
    print(i,end="")
    if i == end_node:
        print("->"+start_node)
        break
    print("->",end="")
```

Output:

```
C:\Users\gallery\Desktop\IS>python Astar.py
The graph is :
Enter the start node for searching : A
Enter the end node for searching : B
A->C->D->B->A
```



```
graph TD
    A ---|100| B
    A ---|20| C
    B ---|20| D
    C ---|20| D
```

Program No.9

Write a program to solve 8 queens problem.

```
/* C/C++ program to solve 8 Queen Problem using backtracking */
#define N 8
#include<stdio.h>
#include<stdbool.h>
/* A utility function to print solution */
void printSolution(int board[N][N])
{
    int i,j;
    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
            printf(" %d ", board[i][j]);
        printf("\n");
    }
}
/* A utility function to check if a queen can be placed on board[row][col]. Note that this function is called
when "col" queens are already placed in columns from 0 to col -1. So we need to check only left side for
attacking queens */
bool isSafe(int board[N][N], int row, int col)
{
    int i, j;
    /* Check this row on left side */
    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;
    /* Check upper diagonal on left side */
    for (i=row, j=col; i>=0 && j>=0; i--, j--)
        if (board[i][j])
            return false;
    /* Check lower diagonal on left side */
    for (i=row, j=col; j>=0 && i<N; i++, j--)
        if (board[i][j])
            return false;
    return true;
}
/* A recursive utility function to solve N Queen problem */
bool solveNQUtil(int board[N][N], int col)
{
    int i;
    /* base case: If all queens are placed then return true */
    if (col >= N)
        return true;
    /* Consider this column and try placing this queen in all rows one by one */
    for (i = 0; i < N; i++)
```

```

{
    /* Check if the queen can be placed on board[i][col] */
    if ( isSafe(board, i, col) )
    {
        /* Place this queen in board[i][col] */
        board[i][col] = 1;
        /* recur to place rest of the queens */
        if ( solveNQUtil(board, col + 1) )
            return true;
        /* If placing queen in board[i][col] doesn't lead to a solution, then remove queen
        from board[i][col] */
        board[i][col] = 0; // BACKTRACK
    }
}
/* If the queen cannot be placed in any row in this column col then return false */
return false;
}
/* This function solves the N Queen problem using Backtracking. It mainly uses solveNQUtil() to
solve the problem. It returns false if queens cannot be placed, otherwise, return true and prints placement
of queens in the form of 1s. Please note that there may be more than one solutions, this function prints one
of the feasible solutions.*/
bool solveNQ()
{
    int board[N][N] = { {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0},
                        {0, 0, 0, 0, 0, 0, 0, 0} };
    if ( solveNQUtil(board, 0) == false )
    {
        printf("Solution does not exist");
        return false;
    }
    printSolution(board);
    return true;
}
// driver program to test above function
int main()
{
    printf("\t8 Queen's Problem\nBelow Matrix represents one solution\n
    Where '1' represents :\tPresence of Queen \n");
    solveNQ();
    return 0;
}

```

Output:

```
      8 Queen's Problem
Below Matrix represents one solution
Where '1' represents : Presence of Queen
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
```

Program No.10

Write a program to solve travelling salesman problem.

```
#include<iostream>
using namespace std;

int ary[10][10],completed[10],n,cost=0;

void takeInput()
{
    int i,j;
    cout<<"Enter the number of Places: ";
    cin>>n;
    cout<<"\nEnter the Cost Matrix\n";
    cout<<"Note: It's an adjacency matrix with each value representing distance\n";
    for(i=0;i < n;i++)
    {
        cout<<"\nEnter Elements of Row: "<<i+1<<"\n";
        for( j=0;j < n;j++)
            cin>>ary[i][j];
        completed[i]=0;
    }
    cout<<"\n\nThe cost list is:";
    for( i=0;i < n;i++)
    {
        cout<<"\n";
        for(j=0;j < n;j++)
            cout<<"\t"<<ary[i][j];
    }
}

int least(int c)
{
    int i,nc=999;
    int min=999,kmin;
    for(i=0;i < n;i++)
    {
        if((ary[c][i]!=0)&&(completed[i]==0))
            if(ary[c][i]+ary[i][c] < min)
            {
                min=ary[i][0]+ary[c][i];
                kmin=ary[c][i];
                nc=i;
            }
    }
    if(min!=999)
        cost+=kmin;
}
```



```

        return nc;
    }

void mincost(int city)
{
    int i,ncity;
    completed[city]=1;
    cout<<city+1<<"--->";
    ncity=least(city);
    if(ncity==999)
    {
        ncity=0;
        cout<<ncity+1;
        cost+=ary[city][ncity];
        return;
    }
    mincost(ncity);
}

int main()
{
    takeInput();
    cout<<"\n\nThe Path is:\n";
    mincost(0); //passing 0 because starting vertex
    cout<<"\n\nMinimum cost is "<<cost;
    return 0;
}

```

Output:

```
Enter the Cost Matrix
Note: It's an adjacency matrix with each value representing distance

Enter Elements of Row: 1
1 2 3 4

Enter Elements of Row: 2
5 6 7 8

Enter Elements of Row: 3
3 4 5 6

Enter Elements of Row: 4
9 8 4 3

The cost list is:
    1      2      3      4
    5      6      7      8
    3      4      5      6
    9      8      4      3

The Path is:
1--->3--->2--->4--->1

Minimum cost is 24
```