Department of Computer Science and Engineering

INDEX

PRACTICAL NO.1

OBJECTIVE: Study of Prolog.

PROLOG-PROGRAMMING IN LOGIC

PROLOG stands for Programming, In Logic — an idea that emerged in the early 1970’s to use logic as programming language. The early developers of this idea included Robert Kowaiski at Edinburgh (on the theoretical side), Marrten van Emden at Edinburgh (experimental demonstration) and Alian Colmerauer at Marseilles (implementation).
David D.H. Warren’s efficient implementation at Edinburgh in the mid -1970’s greatly contributed to the popularity of PROLOG. PROLOG is a programming language centred around a small set of basic mechanisms, Including pattern matching, tree based data structuring and automatic backtracking. This Small set constitutes a surprisingly powerful and flexible programming framework. PROLOG is especially well suited for problems that involve objects- in particular, structured objects- and relations between them.

SYMBOLIC LANGUAGE

PROLOG is a programming language for symbolic, non-numeric computation. It is especially well suited for solving problems that involve objects and relations between objects. For example, it is an easy exercise in prolog to express spatial relationship between objects, such as the blue sphere is behind the green one. It is also easy to state a more general rule: if object X is closer to the observer than object Y. and object Y is closer than Z, then X must be closer than Z. PROLOG can reason about the spatial relationships and their consistency with respect to the general rule. Features like this make PROLOG a powerful language for ArtJIcia1 LanguageA1,) and non- numerical programming.
There are well-known examples of symbolic computation whose implementation in other standard languages took tens of pages of indigestible code, when the same algorithms were implemented in PROLOG, the result was a crystal-clear program easily fitting on one page.

FACTS, RULES AND QUERIES

Progmmming in PROIOG is accomplished by creating a database of facts and rules about objects, their properties, and their relationships to other objects. Queries then can be posed about the objects and valid conclusions will be determined and returned by the program Responses to user queries are determined through a form of inference control known as resolution.
FOR EXAIPLE:
a) FACTS:
Some facts about family relationships could be written as:

```
sister( sue,bill)
parent( ann.sam)
male(jo)
female( riya)
```

b) RULES:

To represent the general rule for grandfather, we write:

```
grand f.gher( X2)
parent(X,Y)
parent( Y,Z)
male(X)
```

c) QUERIES:

Given a database of facts and rules such as that above, we may make queries by typing after a query a symbol'?' statements such as:

```
?-parent(X,sam) Xann
?grandfather(X,Y)
X=jo, Y=sam
```

PROLOG IN DISGINING EXPERT SYSTEMS

An expert system is a set of programs that manipulates encoded knowledge to solve problems in a specialized domain that normally requires human expertise. An expert system's knowledge is obtained from expert sources such as texts, journal articles. databases etc and encoded in a form suitable for the system to use in its inference or reasoning processes. Once a sufficient body of expert knowledge has been acquired, it must be encoded in some form, loaded into knowledge base, then tested, and refined continually throughout the life of the system PROLOG serves as a powerful language in designing expert systems because of its following features.

- Use of knowledge rather than data
- Modification of the knowledge base without recompilation of the control programs.
- Capable of explaining conclusion.
- Symbolic computations resembling manipulations of natural language.
- Reason with meta-knowledge.

META PROGRAMMING

A meta-program is a program that takes other programs as data. Interpreters and compilers are examples of mela-programs. Meta-interpreter is a particular kind of meta-program: an interpreter for a language written in that language. So a PROLOG interpreter is an interpreter for PROLOG, itself written in PROLOG. Due to its symbol- manipulation capabilities, PROLOG is a powerful language for meta-programming. Therefore, it is often used as an implementation language for other languages. PROLOG is particularly suitable as a language for rapid prototyping where we are interested in implementing new ideas quickly. New ideas are rapidly implemented and experimented with.

OUTCOME: Students will get the basic idea of how to program in prolog and its working environment.

PRACTICAL NO.2

OBJECTIVE: Write simple fact for following:

a. Ram likes mango.
b. Seema is a girl.
c. Bill likes Cindy.
d. Rose is red.
e. John owns gold.

Program:

```
Clauses
likes(ram ,mango).
girl(seema).
red(rose).
likes(bill ,cindy).
owns(john ,gold).
```

Output:

```
Goal
queries
?-likes(ram,What).
What= mango
?-likes(Who,cindy).
Who= cindy
?-red(What).
What= rose
?-owns(Who,What).
Who= john
What= gold.
```

OUTCOME: Student will understand how to write simple facts using prolog.

PRACTICAL NO.3

OBJECTIVE: Write predicates One converts centigrade temperatures to Fahrenheit, the other checks if a temperature is below freezing.

Program:
Production rules:
Arithmetic:

$$\begin{tikzcd} \text{c\_to\_f} \arrow[r] & \text{f is c * 9 / 5 +32} \\ \text{freezing} \arrow[r] & \text{f < = 32} \end{tikzcd}$$

Rules:

```
c_to_f(C,F) :-
F is C * 9 / 5 + 32.
freezing(F) :-
F =< 32.
```

Output:
Queries:

```
?- c_to_f(100,X).
X = 212
Yes
?- freezing(15)
.Yes
?- freezing(45).
No
```

OUTCOME: Student will understand how to write a program using the rules.

PRACTICAL NO.4

OBJECTIVE: Write a program to solve the Monkey Banana problem.

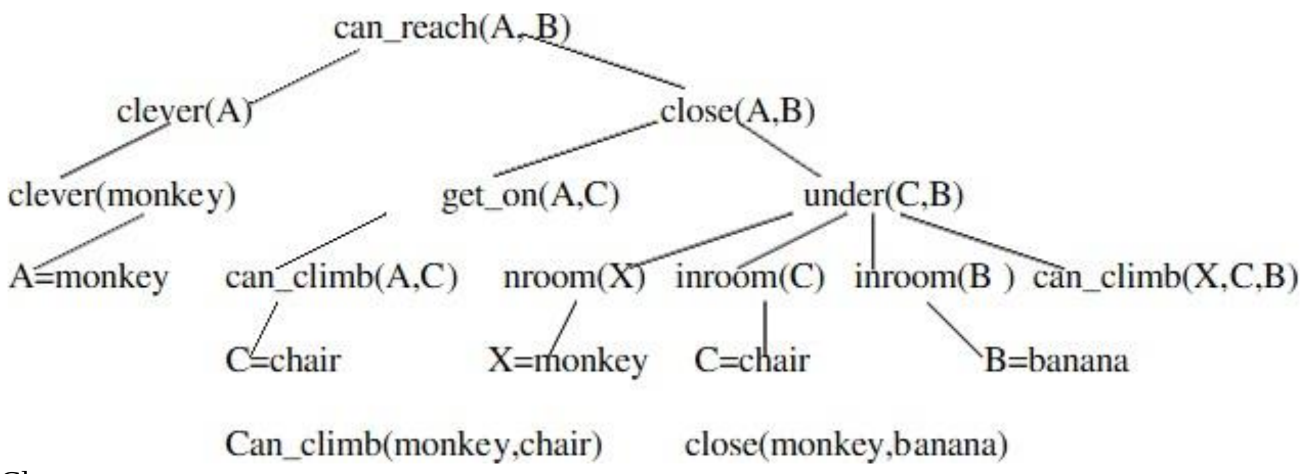
Imagine a room containing a monkey, chair and some bananas. That have been hanged from the centre of ceiling. If the monkey is clever enough he can reach the bananas by placing the chair directly below the bananas and climb on the chair .The problem is to prove the monkey can reach the bananas.The monkey wants it, but cannot jump high enough from the floor. At the window of the room there is a box that the monkey can use. The monkey can perform the following actions:-

1) Walk on the floor.
2) Climb the box.
3) Push the box around (if it is beside the box).
4) Grasp the banana if it is standing on the box directly under the banana.

Production Rules

can_reach $\longrightarrow$ clever,close.
get_on: $\longrightarrow$ can_climb.
under $\longrightarrow$ in room,in_room, in_room,can_climb.
Close $\longrightarrow$ get_on,under| tall

Parse Tree



Clauses:

in_room(bananas).
in_room(chair).
in_room(monkey).
clever(monkey).
can_climb(monkey, chair).
tall(chair).
can_move(monkey, chair, bananas).

```
can_reach(X, Y):-
clever(X),close(X, Y).
get_on(X,Y):- can_climb(X,Y).
under(Y,Z):-
in_room(X),in_room(Y),in_room(Z),can_climb(X,Y,Z).
close(X,Z):-get_on(X,Y),
under(Y,Z);
tall(Y).
```

Output:

```
Queries:
?- can_reach(A, B).
A = monkey.
B = banana.
?- can_reach(monkey, banana).Yes.
```

OUTCOME: Student will understand how to solve monkey banana problem using rules in prolog.

PRACTICAL NO.5

OBJECTIVE: WAP in turbo prolog for medical diagnosis and show t he advantage and disadvantage of green and red cuts.

Program:

```
Domains:
disease,indication=symbol
name-string

Predicates:
hypothesis(name,disease)
symptom(name,indication)
response(char)
go
goonce

clauses
go:-
goonce
write("will you like to try again (y/n)?"),
response(Reply),
Reply='n'.
go.
goonce:-
write("what is the patient's name"),nl,
readln(Patient),
hypothesis(Patient,Disease),!,
write(Patient,"probably has",Disease),!,
goonce:-
write("sorry, i am not ina position to diagnose"),
write("the disease").
symptom(Patient,fever):-
write("does",Patient,"has a fever (y/n)?"),nl,
response(Reply),
Reply='y',nl.
symptom(Patient,rash):-

write ("does", Patient,"has a rash (y/n)?" ),nl,
response(Reply),
Reply='y',
symptom(Patient_body,ache):-
write("does",Patient,"has a body ache (y/n)?"),nl,
response(Reply).
Reply='y',nl.
symptom(Patient,runny_nose):-
write("does",Patient,"has a runny_nose (y/n)?"),
response(Reply),
Reply='y'
hypothesis(Patient,flu):-
symptom(Patient,fever),
symptom(Patient,body_ache),
hypothesis(Patient,common_cold):-
symptom(Patient,body_ache),
Symptom(Patient,runny_nose).
response(Reply):-
readchar(Reply),
write(Reply).
```

Output:

```
makewindow(1,7,7"Expert Medical Diagnosis",2,2,23,70),
go.
```

OUTCOME: Student will understand how to create a expert system using prolog.

PRACTICAL NO.6

OBJECTIVE: WAP to implement factorial, fibonacci of a given number.

Program:

Factorial:

```
factorial(0,1).

factorial(N,F) :-
   N>0,
   N1 is N-1,
   factorial(N1,F1),
   F is N * F1.
```

Output:
Goal:

```
?- factorial(4,X).
X=24
```

Fibonacci:

```
fib(0, 0).
fib(X, Y) :- X > 0, fib(X, Y, _).
fib(1, 1, 0).
fib(X, Y1, Y2) :-
 X > 1,
 X1 is X - 1,
 fib(X1, Y2, Y3),
 Y1 is Y2 + Y3.
```

Output:
Goal:

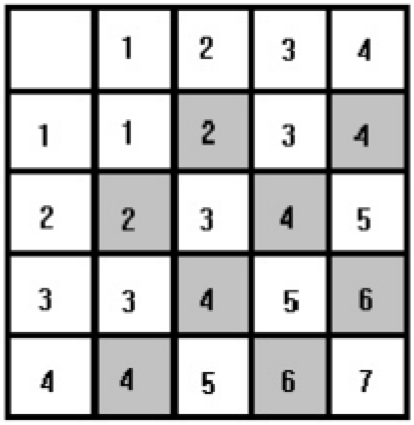
```
?-fib(10,X).
X=55
```

OUTCOME: Student will understand the implementation of Fibonacci and factorial series using prolog.

PRACTICAL NO.7

OBJECTIVE: Write a program to solve 4-Queen problem.

Program:
In the 4 Queens problem the object is to place 4 queens on a chessboard in such a way that no queens can capture a piece. This means that no two queens may be placed on the same row, column, or diagonal.



The n Queens Chessboard.

```
domains
queen = q(integer, integer)
queens = queen*
freelist = integer*
board = board(queens, freelist, freelist, freelist, freelist)
predicates
nondeterm placeN(integer, board, board)
nondeterm place_a_queen(integer, board, board)
nondeterm nqueens(integer)
nondeterm makelist(integer, freelist)
nondeterm findandremove(integer, freelist, freelist)
nextrow(integer, freelist, freelist)
clauses
nqueens(N):-
makelist(N,L),
Diagonal=N*2-1,
makelist(Diagonal,LL),
placeN(N,board([],L,L,LL,LL),Final),
write(Final).
placeN(_,board(D,[],[],D1,D2),board(D,[],[],D1,D2)):-!.
placeN(N,Board1,Result):-
place_a_queen(N,Board1,Board2),
placeN(N,Board2,Result).
place_a_queen(N,
board(Queens,Rows,Columns,Diag1,Diag2),
board([q(R,C)|Queens],NewR,NewC,NewD1,NewD2)):-
nextrow(R,Rows,NewR),
findandremove(C,Columns,NewC),
D1=N+C-R,findandremove(D1,Diag1,NewD1),
D2=R+C-1,findandremove(D2,Diag2,NewD2).
findandremove(X,[X|Rest],Rest).
findandremove(X,[Y|Rest],[Y|Tail]):-
findandremove(X,Rest,Tail).
makelist(1,[1]).
makelist(N,[N|Rest]) :-
N1=N-1,makelist(N1,Rest).
nextrow(Row,[Row|Rest],Rest).
```

Output:

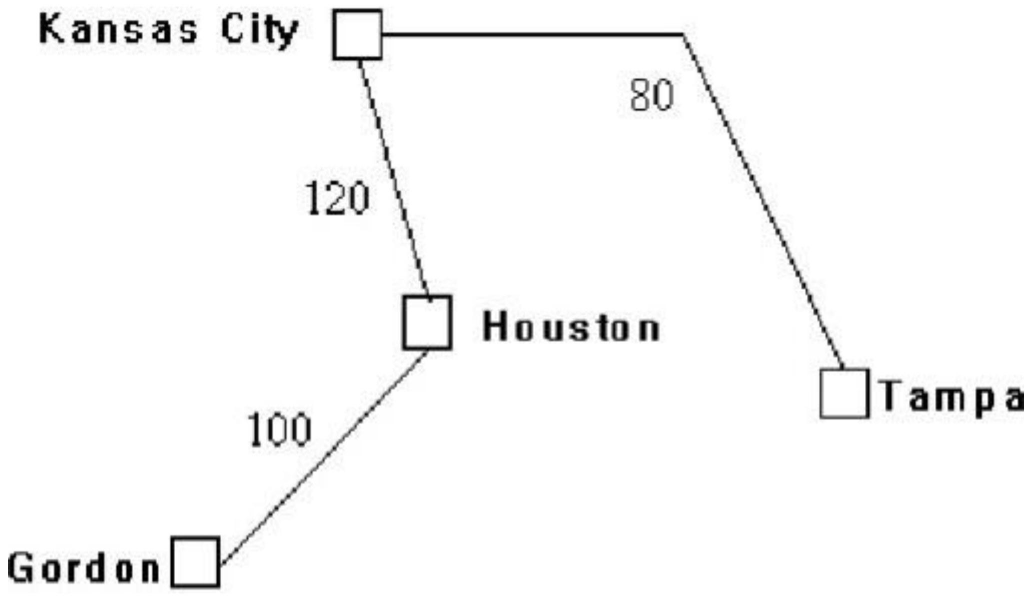
```
Goal:
?-nqueens(4),nl.
board([q(1,2),q(2,4),q(3,1),q(4,3),[],[],[7,4,1],[7,4,1])
yes
```

OUTCOME: Student will implement 4-Queen problem using prolog.

PRACTICAL NO.8

OBJECTIVE: Write a program to solve traveling salesman problem.

The following is the simplified map used for the prototype:



Program:

```
Production Rules:-
route(Town1,Town2,Distance)          road(Town1,Town2,Distance).
route(Town1,Town2,Distance)
          road(Town1,X,Dist1),route(X,Town2,Dist2),Distance=Dist1+Dist2,
domains
town = symbol
distance = integer
predicates
nondeterm road(town,town,distance)
nondeterm route(town,town,distance)

clauses
road("tampa","houston",200).
road("gordon","tampa",300).
road("houston","gordon",100).
road("houston","kansas_city",120).
road("gordon","kansas_city",130).
route(Town1,Town2,Distance):-
road(Town1,Town2,Distance).
route(Town1,Town2,Distance):-
road(Town1,X,Dist1),
route(X,Town2,Dist2),
Distance=Dist1+Dist2,!.
```

Output:

Goal:

```
route("tampa", "kansas_city", X),
write("Distance from Tampa to Kansas City is ",X),nl.

Distance from Tampa to Kansas City is 320
X=320
```

OUTCOME: Student will implement travelling salesmen problem using prolog.

PRACTICAL NO.9

OBJECTIVE: Write a program to solve water jug problem using LISP.

Program:

```
;returns the quantity in first jug
(defun get-first-jug (state) (car state))

;returns the quantity in second jug
(defun get-second-jug (state) (cadr state))

;returns the state of two jugs
(defun get-state (f s) (list f s))

;checks whether a given state is a goal
; GOAL IS TO GET 4 IN SECOND JUG
(defun is-goal (state)
    (eq (get-second-jug state) 4))

;returns all possible states that can be derived
;from a given state
(defun child-states (state)
    (remove-null
 (list
     (fill-first-jug state)
     (fill-second-jug state)
     (pour-first-second state)
     (pour-second-first state)
     (empty-first-jug state)
     (empty-second-jug state))))

;remove the null states
(defun remove-null (x)
    (cond
 ((null x) nil)
 ((null (car x)) (remove-null (cdr x)))
 ((cons (car x) (remove-null (cdr x))))))

;return the state when the first jug is filled (first jug can hold 3)
(defun fill-first-jug (state)
    (cond
 ((< (get-first-jug state) 3) (get-state 3 (get-second-jug state))))))

;returns the state when the second jug is filled (second jug can hold 5)
(defun fill-second-jug (state)
    (cond
 ((< (get-second-jug state) 5) (get-state (get-first-jug state) 5))))

;returns the state when quantity in first
;is poured to second jug
(defun pour-first-second (state)
    (let  (  (f  (get-first-jug  state))
    (s (get-second-jug state)))
 (cond
     ((zerop f) nil) ; first jug is empty
     ((= s 5) nil) ; Second jug is full
     ((<= (+ f s) 5)
  (get-state 0 (+ f s)))
     (t    ; pour to first from second
  (get-state (- (+ f s) 5) 5)))))

;returns the state when second jug is poured to first
(defun pour-second-first (state)
    (let ( (f (get-first-jug state))
      (s (get-second-jug state)))
 (cond
     ((zerop s) nil) ; second jug is empty
     ((= f 3) nil) ; second jug is full
     ((<= (+ f s) 3)
  (get-state (+ f s) 0))
     (t    ;pour to second from first
  (get-state 3 (- (+ f s) 3))))))

;returns the state when first jug is emptied
(defun empty-first-jug (state)
    (cond
 ((> (get-first-jug state) 0) (get-state 0 (get-second-jug state)))))

;returns the state when second jug is emptied
(defun empty-second-jug (state)
    (cond
 ((> (get-second-jug state) 0) (get-state (get-first-jug state) 0))))


 ;;;MAIN FUNCTION
(defun dfs (start-state depth lmt)
    (setf *node* 0)
    (setf *limit* lmt)
    (dfs-node start-state depth)
)
```

```
;dfs-node expands a node and calls dfs-children to recurse on it
(defun dfs-node (state depth)
    (setf *node* (+ 1 *node*))
    (cond
 ((is-goal state) (list state))
 ((zerop depth) nil)
 ((> *node* *limit*) nil)
 ((let ((goal-child (dfs-children (child-states state) (- depth 1))))
   (and goal-child (cons state goal-child)))) ;for back-tracking if the branch don't have a goal
state
 ))

;dfs-children expands each node recursively and give it
;to dfs-node to process
(defun dfs-children (states depth)
    (cond
 ((null states) nil)
 ((dfs-node (car states) depth))
 ((dfs-children (cdr states) depth))))

(print "ENTER YOUR INPUT AS")
(print "(dfs start_state depth limit)")
```

OUTCOME: Student will implement water-jug problem using Lisp.