ABES Institute of Technology Ghaziabad

Affiliated to Dr. A.P.J. AKTU, Lucknow

LAB FILE

Department of Computer Science & Engineering

Subject Name: Artificial Intelligence lab

Subject Code: RCS-751

Session: 2020-2021

Semester: 7th

Submitted To:
Mrs. Aishwarya Gupta

Submitted By:
Student Name - Mihir Luthra
Roll. No. - 1729010091
Section: 4CS-B

INDEX

ABES Institute of Technology, Ghaziabad

Department of Computer Science & Engineering

Institute Vision	To be leading institution in technical education providing education and training enabling human resource to serve nation and world at par with global standards in education
Institute Mission	1)Developing state of art infrastructure which also includes establishment of centre of excellence in pursuit of academic and technical excellence 2)Valuing work force inculcating belongingness and professional integrity 3)To develop human resource to solve local, regional and global problems to make technology relevant to those who mean it most

ABES Institute of Technology, Ghaziabad

Department of Computer Science & Engineering

Department Vision	To provide excellence by imparting knowledge to the learners enabling them to become skilled professionals to be recognized as a responsible citizen.
Department Mission	1) Provide quality education in the field of computer science and engineering through experienced and qualified faculty members. 2) Motivate learners for higher studies and research oriented activities by utilizing resources of Centers of Excellence. 3) Inculcate societal values, professional ethics, team work, and leadership qualities by having exposure at National and International level activities.

ABES Institute of Technology, Ghaziabad

Department of Computer Science & Engineering

Program Educational Objectives (PEOs)

PEO 1	Graduates of the program are expected to be employed in IT industry or Indulge in higher studies and research.
PEO 2	Graduates of the program are expected to exhibit curiosity to learn new technologies and work with ethical values and team work.
PEO 3	Graduates of the program are expected to design and develop innovative solutions related to real world problems of the society.

Program Specific Outcomes (PSOs)

PSO 1	Solve complex problems using data structures and other advanced suitable algorithms.
PSO 2	Interpret fundamental concepts of computer systems and understand its hardware and software aspect.
PSO 3	Analyze the constraints of the existing data base management systems and get experience on large-scale analytical methods in the evolving technologies.
PSO 4	Develop intelligent systems and implement solutions to cater the business specific requirements.

List of Programs

1. Study of Prolog.
2. Write simple fact for the statements using PROLOG.
3. Write predicates One converts centigrade temperatures to Fahrenheit, the other checks if a temperature is below freezing.
4. Write a program to solve the Monkey Banana problem.
5. WAP in turbo prolog for medical diagnosis and show the advantage and disadvantage of green and red cuts.
6. WAP to implement factorial, fibonacci of a given number.
7. Write a program to solve 4-Queen problem.
8. Write a program to solve traveling salesman problem.
9. Write a program to solve water jug problem using LISP
10. Write a Lisp program to solve best first search traversal.

PROGRAM 1

STUDY OF PROLOG

1.1 Study of Prolog

The Prolog language allows us to explore a wide range of topics in discrete mathematics, logic, and computability. Prolog's powerful pattern-matching ability and its computation rule give us the ability to experiment in two directions. For example, a typical experiment might require a test of a definition with a few example computations Prolog computation rule also allows a definition to be tested in reverse, by specifying a result and then asking for the elements that give the result.

Following are feature of prolog programming language

1. In purely declarative languages, the programmer only states what the problem is and leaves the rest to the language system.
2. We'll see specific, simple examples of cases where Prolog fits really well shortly.
3. Prolog = Programmation en Logique (Programming in Logic).
4. In a declarative language the programmer specifies a goal to be achieved the Prolog system works out how to achieve it
5. Relational databases owe something to Prolog
6. Prolog (programming in logic) is a logic-based programming language: programs correspond to sets of logical formulas and the Prolog interpreter uses logical methods to resolve queries.
7. Prolog is a declarative language: you specify what problem you want to solve rather than how to solve it.
8. Prolog is very useful in some problem areas, such as artificial intelligence, natural language processing, databases, but pretty useless in others, such as for instance graphics

Applications of Prolog:

1. Intelligent database retrieval
2. Natural language understanding
3. Expert systems
4. Specification language
5. Machine learning
6. Robot planning
7. Automated reasoning
8. Problem solving

To start the Prolog interpreter in a UBUNTU environment type and hit return. Once Prolog has started up it displays the prompt

```
|?-
```

which indicates that the interpreter is waiting for a command from the user. All commands must end with a period. For example, the command

```
|?- integer(3.4).
```

returns the answer no because 3.4 is not an integer. A command is usually called a ***goal*** **or a** ***query.*** A Prolog program is a set of facts or rules called *definite clauses*.

Loading Information

To read in the contents of a file named ***filename*** type

|?- [filename].

and hit return. If the file name contains characters other than letters or digits, then put single quotes around the filename. For example, if the name of the file **is *file.p*,** then type

|?- ‘file.p’].

will load the file named *file.p*.

You can read in several files at once. For example, to read in files named *foo*, *goo*, and *moo* type

|?- [foo, goo, moo].

Sometimes it may be useful to enter a few clauses directly from a terminal to test something or other. In this case you must type the command

|?- [user].

and hit return. The prompt will appear to indicate that the interpreter is waiting for data.

Variables, Predicates, and Clauses

Variables

A variable may be represented by a string of characters made up of letters or digits or the underscore symbol _, and that begins with either an uppercase letter or _. For example, the following strings denote variables:

X, Input_list, Answer, _,

A variable name that begins with the underscore symbol represents an unspecified (or anonymous) variable.

Predicates

A *predicate* is a relation. In Prolog the name of a predicate is an alphanumeric string of characters (including _) that begins with a lowercase letter.

Clauses

The power of Prolog comes from its ability to process clauses. A *clause* is a statement taking one of the forms head.or head :- body. where the head is an atomic formula and the body is a sequence of atomic formulas separated by commas. For example, the following statements are clauses.

capital–of(salem, oregon).

q(b).

p(X) :- q(X), r(X), s(X).

The last clause has head p(X) and its body consists of the atomic formulas

q(X), r(X), and s(X).

The meaning of a clause of the form head is that head is true. A clause of the form

head :- body.

has a declarative meaning and a procedural meaning. The declarative meaning is that the head is true if all of the atomic formulas in the body are true. The procedural meaning is that for the head to succeed, each atomic formula in the body must succeed. For example, suppose we have the clause

p(X) :- q(X), r(X), s(X).

From the declarative point of view this means that for all X, p(X) is true if q(X) and r(X) and s(X) are true. From the procedural point of view this means that for all X, p(X) succeeds if q(X), r(X), and s(X) succeed.

Clauses

Prolog also allows us to represent the “or” operation in two different ways. For example, suppose that we have the following two Prolog clauses to define the parentof relation in terms of the mother of and father of relations.

parentof(X, Y) :- motherof(X, Y).

parentof(X, Y) :- fatherof(X, Y).

Unification

Unification is the process of matching two expressions by attempting to construct a set of bindings for the variables so that when the bindings are applied to the two expressions, they become syntactically equal. Unification is used as part of Prolog's computation rule. The following symbol is used for unification within a program.= If two expressions can be unified, then Prolog will return with corresponding bindings for any variables that occur in the expressions. For example, try out the following tests.

```
|?- b = b.
|?- p(a) = p(a).
|?- p(X) = p(b).
|?- 5 = 5.
|?- 2 + 3 = 1 + 4.
```

Computation

A Prolog program executes goals, where a *goal* is the body of a clause. In other words, a goal is one or more atomic formulas separated by commas. The atomic formulas in a goal are called *subgoals*. For example, the following expression is a goal consisting of two subgoals.

```
|?- par(X, james), par(Y, X).
```

The execution of a goal proceeds by unifying the subgoals with heads of clauses. The search for a matching head starts by examining clauses at the beginning of the program and proceeds linearly through the clauses. If there are two or more subgoals, then they are executed from left to right. A subgoal is true in two cases:

1. It matches a fact (i.e., the head of a bodyless clause).
2. It matches the head of a clause with a body and when the matching

substitution is applied to the body, each subgoal of the body is true. A goal is true if there is a substitution that when applied to its subgoals makes each subgoal true. For example, suppose we have the following goal for the introductory program example.

```
|?- par(X, james), par(Y, X).
```

This goal is true because there is a substitution {X=ruth, Y=katherine} that when applied to the two subgoals gives the following subgoals, both of which are true. par(ruth, james), par(katherine, ruth).

Experiments to Perform

1. Try out some unification experiments like the following. First find the answers by hand. Then check your answers with Prolog.

```
|?- p(X) = p(a).
|?- p(X, f(Y)) = p(a, Z).
|?- p(X, a, Y) = p(b, Y, Z).
|?- p(X, f(Y, a), Y) = p(f(a, b), V, Z).
|?- p(f(X, g(Y)), Y) = p(f(g(a), Z), b).
```

Numeric Computations

Prolog has a built-in predicate "is" that is used to evaluate numerical expressions. The predicate is infix with a variable on the left and a numerical expression on the right. For example, try out the following goals.

```
|?- X is 5 + 7.
|?- X is 5 - 4 + 2.
|?- X is 5 * 45.
|?- X is log(2.7).
|?- X is exp(1).
|?- X is 12 mod 5.
```

The expression on the right must be able to be evaluated. For example, try out the goal

```
|?- X is Y + 1.
```

Now try out the goal

```
|?- Y is 5, X is Y + 1.
```

SICStus Prolog has a rich set of numerical operations that includes all of the ISO operations:
Binary operators +, –, *, /, //, rem, mod, sin, cos, atan. Unary operators +, –, abs, ceiling, floor, float, truncate, round, exp, sqrt, log.

Numeric Comparison.

Numerical expressions can be compared with binary comparison operators. The six operators are given as follows:
=:=, =\=, <, >, =<, >=.
For example, try out the following goals.

```
|?- 5 < 6 + 2.
|?- 5 =< 6 + 2.
|?- 5 =:= 6 –
```

Family Trees
In this experiment we'll continue working with a family tree by examining a few of the many family relations. To keep things short and concise, let p(X, Y) mean that X is a parent of Y, and let g(X, Y) mean that X is a grandparent of Y.

Outcome - *Students study logic programming language Prolog*.

PROGRAM 2

WRITE SIMPLE FACT FOR THE STATEMENTS USING PROLOG

Experiment:-

1. ram likes apple
2. sita likes rain
3. Everybody likes god
4. gold is valuable
5. she likes dolls but hates dogs

Query:-

1. sita likes gita
2. is silver valuable
3. priya Likes god
4. shreya likes dolls

Program:-

```
likes(ram,apple).

likes(sita,rain).

likes(_).

valuable(gold).

likes(_,dolls,dogs).
```

Output:-

```
| ?- likes(sita,gita).
no

| ?- valuable(silver).
 no

| ?- likes(priya).
yes

| ?- likes(shreya,dolls,dogs).
yes

| ?-
```

PROGRAM 3

Write predicates that can converts centigrade temperatures to Fahrenheit, the other checks if a temperature is below freezing

Formula for Centigrade (C) temperatures to Fahrenheit (F) :-

F = C * 9 / 5 + 32

Rule:-

Centigrade to Fahrenheit (c_to_f)F is C * 9 / 5 + 32

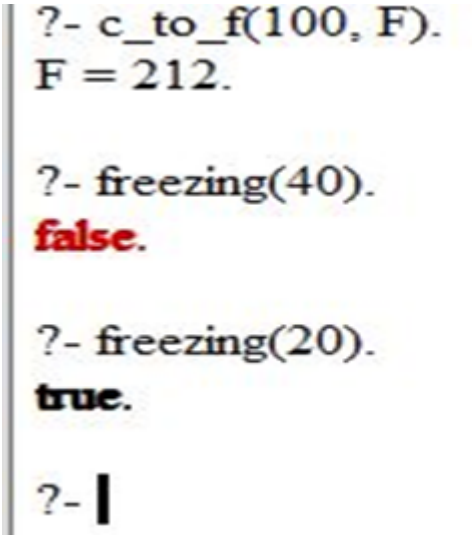
Program:-

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

% here freezing point is less than 32 Fahrenheit

freezing (F) :-F =< 32.
```

OUTPUT:-



PROGRAM 4

Write a program to solve Monkey Banana Problem.

Program:-

```
on(floor,monkey).

on(floor,box).

in(room,monkey).

in(room,box).

in(room,banana).

at(ceiling,banana).

strong(monkey).

grasp(monkey).

climb(monkey,box).

push(monkey,box):-strong(monkey).

under(banana,box):-push(monkey,box).

canreach(banana,monkey):-at(floor,banana);
at(ceiling,banana),under(banana,box),climb(monkey,box).

canget(banana,monkey):-canreach(banana,monkey),grasp(monkey).
```

OUTPUT:-

```
?- canreach(banana , monkey).
true.

?- trace
|    .
true.

[trace]  ?- canget(banana, monkey).
   Call: (8) canget(banana, monkey) ? creep
   Call: (9) canreach(banana, monkey) ? creep
   Call: (10) at(floor, banana) ? creep
   Fail: (10) at(floor, banana) ? creep
   Redo: (9) canreach(banana, monkey) ? creep
   Call: (10) at(ceiling, banana) ? creep
   Exit: (10) at(ceiling, banana) ? creep
   Call: (10) under(banana, box) ? creep
   Call: (11) push(monkey, box) ? creep
   Call: (12) strong(monkey) ? creep
   Exit: (12) strong(monkey) ? creep
   Exit: (11) push(monkey, box) ? creep
   Exit: (10) under(banana, box) ? creep
   Call: (10) climb(monkey, box) ? creep
   Exit: (10) climb(monkey, box) ? creep
   Exit: (9) canreach(banana, monkey) ? creep
   Call: (9) grasp(monkey) ? creep
   Exit: (9) grasp(monkey) ? creep
   Exit: (8) canget(banana, monkey) ? creep
true.
```

PROGRAM 5

WAP in turbo prolog for medical diagnosis

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 in a 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,

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.
```

PROGRAM 6

WAP to implement factorial and 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:

```
?- 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:

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

Program-7

WAP to solve the N-Queen Problem.

PROGRAM

```
queens (N, Queens) :-
length(Queens, N),
board(Queens, Board, 0, N, _ , _ ),
queens(Board, 0, Queens).
board([], [] , N, N, _, _ ).
board([_] Queens],[Col-varsl Board], Col0, N, [_IVR], VC) :-
Col is  Col0+1,
function (Vars, f, N),
constraints(N, Vars, VR, VC).
board (Queens, Board, Col, N, VR, [_IVC]).
constraints(0, _, _, _ ) ;- !.
constraints(N, Row, [RRs], [ClCs]):-
arg(N, Row, R-C),
 M is N-1,
constraints (M, Row, Rs, Cs).
queens ([], _,[]).
queens ([ClCs), Row0, [CollSolution]) :-
Row is Row0+1,
 select(Col-Vars, [ClCs), Board),
arg(Row, Vars, Row-Row),
Queens (Board, Row, Solution).
```

Program-8

Write a program to implement the Travelling Salesman Problem.

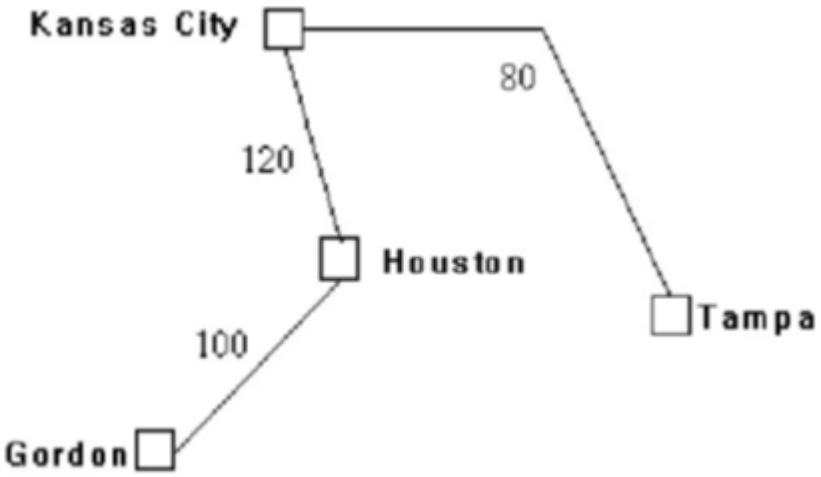


Fig : simplified map used for the prototype.

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
```

Program-9

Write a program to solve the Water Jug Problem.

PROGRAM

```
(defvar state);declare global variable state
(defun start ()
(data) (checkB))
(defun data()
(
format t "~%WATER JUG PROBLEM")
(format t "~%FILL JUGNAME: (fillJUGNAME)")
(format t "~%EMPTY JUGNAME: (emptyJUGNAME)")
(format t "~%PUT A TO B : (putAtoB)")
(format t "%%GIVE THE STARTING STATE OF EACH JUG IN FORM (A B C) E.G (0 0 0)")
(format t "~%MAX A=7,MAX B=4 AND MAX C=2: ")
(setf state (read));SET VALUE AT VARIABLE STATE
)
(format t "~%STARTING CONDITION OF EACH JUG: ~3a" state)
;FILLING A
(defun fillA () (let((x(first state))(y(second state))(z(third state)))
(if (< x 7) (let ((newstate (list 7 y z)))
(prog1 state (format t "~%NEW STATE: ~3a" newstate ))
(setf (first state) 7)))
(if (>= x 7) (format t "~%A IS ALREADY FULL")))
(checkB))
;FILLING B
(defun fillB () (let((x(first state))(y(second state))(z(third state)))
```

```
(if (< y 4) (let ((newstate (list x 4 z)))
(prog1 state (format t "~%NEW STATE: ~3a" newstate ))
(setf (second state) 4)))
(if (>= y 4)(format t "~%B IS ALREADY FULL")))
(checkB))
;FILLING C
(defun fillC () (let((x(first state))(y(second state))(z(third state)))
(if (< z 2) (let ((newstate (list x y 2)))
(prog1 state (format t "~%NEW STATE: ~3a" newstate ))
(setf (third state) 2)))
(if (>= z 2) (format t "~%C IS ALREADY FULL")))
(checkB) )
;EMPTY A
(defun emptyA () (let((x(first state))(y(second state))(z(third state)))
(if (> x 0) (let ((newstate (list 0 y z)))
(prog1 state (format t "~%NEW STATE: ~3a" newstate ))
(setf (first state) 0)))
(if (<= x 0) (format t "~%A IS ALREADY EMPTY")))
(checkB))
;EMPTY B
(defun emptyB () (let((x(first state))(y(second state))(z(third state)))
(if (> y 0) (let ((newstate (list x 0 z)))
(prog1 state (format t "~%NEW STATE: ~3a" newstate ))
(setf (second state) 0)))
(if (<= y 0) (format t "~%B IS ALREADY EMPTY")))
(checkB))
;EMPTY C
(defun emptyC () (let((x(first state))(y(second state))(z(third state)))
(if (> z 0) (let ((newstate (list x y 0)))
```

```
(prog1 state (format t "~%NEW STATE: ~3a" newstate ))
(setf (third state) 0)))
(if (<= z 0) (format t "~%C IS ALREADY EMPTY")))
(checkB))

;PUT A TO B
(defun putAtoB () (let((x(first state))(y(second state))(z(third state)))
(cond((<= x 0) (format t"~%A IS ALREADY EMPTY"))
((>= y 4) (format t"~%B IS FULL"))
((and (> x 0) (< y 4))
(let ((local (- 4 y)))
(if (or (> x local)(= x local))
(let ((newstate (list (- x (- 4 y)) 4 z)))
(setf (first state) (first newstate))
(setf (second state) (second newstate))
(setf (third state) (third newstate))
(format t "~%NEW STATE: ~3a" newstate))
(let ((newstate (list 0 (+ y x) z)))
(setf (first state) (first newstate))
(setf (second state) (second newstate))
(setf (third state) (third newstate))
(format t "~%NEW STATE: ~3a" newstate))
)))))
(checkB))
;PUT B TO A
(defun putBtoA () (let((x(first state))(y(second state))(z(third state)))
(cond((>= x 7) (format t"~%A IS FULL"))
((<= y 0) (format t"~%B IS EMPTY"))
((and (> y 0) (< x 7))
```

```
(let ((local (- 7 x)))
(if (or (> x local)(= x local))
(let ((newstate (list 7 (- y (- 7 x)) z)))
(setf (first state) (first newstate))
(setf (second state) (second newstate))
(setf (third state) (third newstate))
(format t "~%NEW STATE: ~3a" newstate))
(let ((newstate (list (+ y x) 0 z)))
(setf (first state) (first newstate))
(setf (second state) (second newstate))
(setf (third state) (third newstate))
(format t "~%NEW STATE: ~3a" newstate))
)))))
(checkB))
;PUT A TO C
(defun putAtoC () (let((x(first state))(y(second state))(z(third state)))
(cond((<= x 0) (format t"~%A IS EMPTY"))
((>= z 2) (format t"~%C IS FULL"))
((and (> x 0) (< z 2))
(let ((local (- 2 z)))
(if (or (> x local)(= x local))
(let ((newstate (list (- x (- 2 z)) y 2)))
(setf (first state) (first newstate))
(setf (second state) (second newstate))
(setf (third state) (third newstate))
(format t "~%NEW STATE: ~3a" newstate))
(let ((newstate (list 0 y (+ z x))))
(setf (first state) (first newstate))
(setf (second state) (second newstate))
```

```
(setf (third state) (third newstate))
(format t "~%NEW STATE: ~3a" newstate))
)))))
(checkB))
;PUT C TO A
(defun putCtoA () (let((x(first state))(y(second state))(z(third state)))
(cond((>= x 7) (format t"~%A IS FULL"))
((<= z 0) (format t"~%C IS EMPTY"))
((and (> z 0) (< x 7))
(let ((local (- 7 x)))
(if (or (> x local)(= x local))
(let ((newstate (list 7 y (- z (- 7 x)))))
(setf (first state) (first newstate))
(setf (second state) (second newstate))
(setf (third state) (third newstate))
(format t "~%NEW STATE: ~3a" newstate))
(let ((newstate (list (+ z x) y 0)))
(setf (first state) (first newstate))
(setf (second state) (second newstate))
(setf (third state) (third newstate))
(format t "~%NEW STATE: ~3a" newstate))
)))))
(checkB))
;PUT B TO C
(defun putBtoC () (let((x(first state))(y(second state))(z(third state)))
(cond((<= y 0) (format t"~%B IS EMPTY"))
((>= z 2) (format t"~%C IS FULL"))
((and (> y 0) (< z 2))
(let ((local (- 2 z)))
```

```
(if (or (> y local)(= y local))
(let ((newstate (list x (- y (- 2 z)) 2)))
(setf (first state) (first newstate))
(setf (second state) (second newstate))
(setf (third state) (third newstate))
(format t "~%NEW STATE: ~3a" newstate))
(let ((newstate (list x 0 (+ z y))))
(setf (first state) (first newstate))
(setf (second state) (second newstate))
(setf (third state) (third newstate))
(format t "~%NEW STATE: ~3a" newstate))
)))))
(checkB))
;PUT C TO B
(defun putCtoB () (let((x(first state))(y(second state))(z(third state)))
(cond((>= y 4) (format t"~%B IS FULL"))
((<= z 0) (format t"~%C IS EMPTY"))
((and (> z 0) (< y 4))
(let ((local (- 4 y)))
(if (or (> y local)(= y local))
(let ((newstate (list x 4 (- z (- 4 y)))))
(setf (first state) (first newstate))
(setf (second state) (second newstate))
(setf (third state) (third newstate))
(format t "~%NEW STATE: ~3a" newstate))
(let ((newstate (list x (+ y z) 0)))
(setf (first state) (first newstate))
(setf (second state) (second newstate))
(setf (third state) (third newstate))
```

```
(format t "~%NEW STATE: ~3a" newstate))

)))))

(checkB))

;CHECK IF B=1

(defun checkB () (let ((y(second state)))

(if (= y 1) (format t "~% YOU WIN B=1."

))))
```

OUTPUT