CIE\_3

3A)

Code:

# 3a: Defining facts and rules in a simple way

# Facts

Cat = {"Tom"} # Set of cats

Mary\_allergic\_to\_cats = True # Mary is allergic to cats

LivesWith\_Mary\_and\_Cat = True # We assume Mary lives with a cat (as per the context)

Allergic = {"Mary"} # Set of people who suffer from allergies (we'll start with Mary)

# Rule: If someone suffers from allergies, they sneeze

def sneeze(x):

return x in Allergic

# Rule: If someone lives with a cat and is allergic to it, then they suffer from allergies

def suffer\_allergies(x):

if LivesWith\_Mary\_and\_Cat and Mary\_allergic\_to\_cats:

Allergic.add("Mary")

# Apply the rule to see if Mary sneezes

suffer\_allergies("Mary")

# Now, check if Mary sneezes

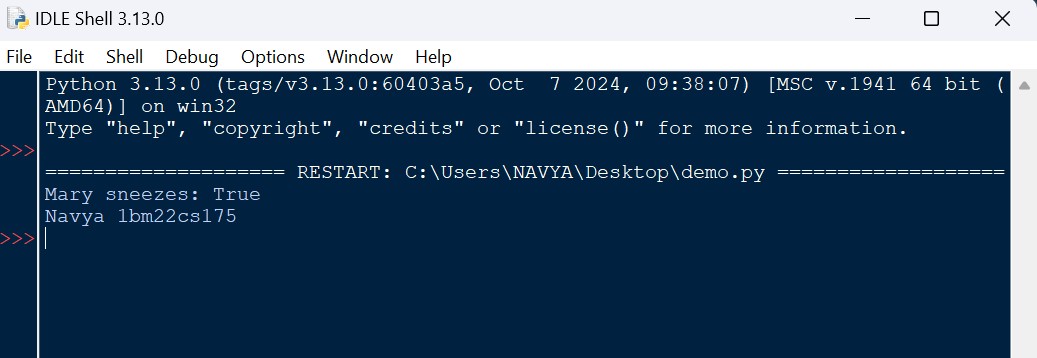
if sneeze("Mary"):

print("Mary sneezes: True")

else:

print("Mary sneezes: False")

output:



3B)

Code:

# Define basic classes for FOL terms, predicates, and quantifiers

class Term:

pass

class Variable(Term):

def \_\_init\_\_(self, name):

self.name = name

def \_\_repr\_\_(self):

return self.name

class Function(Term):

def \_\_init\_\_(self, func\_name, \*args):

self.func\_name = func\_name

self.args = args

def \_\_repr\_\_(self):

return f"{self.func\_name}({', '.join(map(str, self.args))})"

class Predicate:

def \_\_init\_\_(self, pred\_name, \*args):

self.pred\_name = pred\_name

self.args = args

def \_\_repr\_\_(self):

return f"{self.pred\_name}({', '.join(map(str, self.args))})"

# Define logical operations for Predicate

def \_\_and\_\_(self, other):

if isinstance(other, Predicate):

return Conjunction(self, other)

return NotImplemented

def \_\_or\_\_(self, other):

if isinstance(other, Predicate):

return Disjunction(self, other)

return NotImplemented

def \_\_invert\_\_(self):

return Negation(self)

def \_\_rshift\_\_(self, other):

if isinstance(other, Predicate):

return Implication(self, other)

return NotImplemented

class Quantifier:

def \_\_init\_\_(self, quantifier, variable, expression):

self.quantifier = quantifier # 'forall' or 'exists'

self.variable = variable

self.expression = expression

def \_\_repr\_\_(self):

return f"{self.quantifier} {self.variable} ({self.expression})"

# Logical Connective Classes

class Conjunction:

def \_\_init\_\_(self, left, right):

self.left = left

self.right = right

def \_\_repr\_\_(self):

return f"({self.left} & {self.right})"

class Disjunction:

def \_\_init\_\_(self, left, right):

self.left = left

self.right = right

def \_\_repr\_\_(self):

return f"({self.left} | {self.right})"

class Negation:

def \_\_init\_\_(self, expression):

self.expression = expression

def \_\_repr\_\_(self):

return f"~({self.expression})"

class Implication:

def \_\_init\_\_(self, left, right):

self.left = left

self.right = right

def \_\_repr\_\_(self):

return f"({self.left} -> {self.right})"

# Helper function to create FOL statements

def forall(variable, expression):

return Quantifier('∀', variable, expression)

def exists(variable, expression):

return Quantifier('∃', variable, expression)

# FOL Representation for all the examples

# i. Every real number has its corresponding negative.

x = Variable('x')

y = Variable('y')

Real = Predicate('Real', x)

negative = Function('-', x)

# Real(x) -> exists y (Real(y) & (y = -(x)))

expression\_i = forall(x, exists(y, Conjunction(Real, Conjunction(Predicate('Real', y),

Predicate('=', y, negative)))))

print("FOL representation i:", expression\_i)

# ii. Everybody loves somebody.

Loves = Predicate('Loves', x, y)

expression\_ii = forall(x, exists(y, Conjunction(Predicate('Person', x),

Conjunction(Predicate('Person', y), Loves))))

print("FOL representation ii:", expression\_ii)

# iii. There is somebody whom no one loves.

expression\_iii = exists(x, forall(y, Implication(Predicate('Person', y), Negation(Loves))))

print("FOL representation iii:", expression\_iii)

# iv. Susan brought everything that Ronald bought.

Bought = Predicate('Bought', 'Ronald', x)

Brought = Predicate('Brought', 'Susan', x)

expression\_iv = forall(x, Implication(Bought, Brought))

print("FOL representation iv:", expression\_iv)

# v. Parrot is green while rabbit is not.

Green = Predicate('Green', 'Parrot')

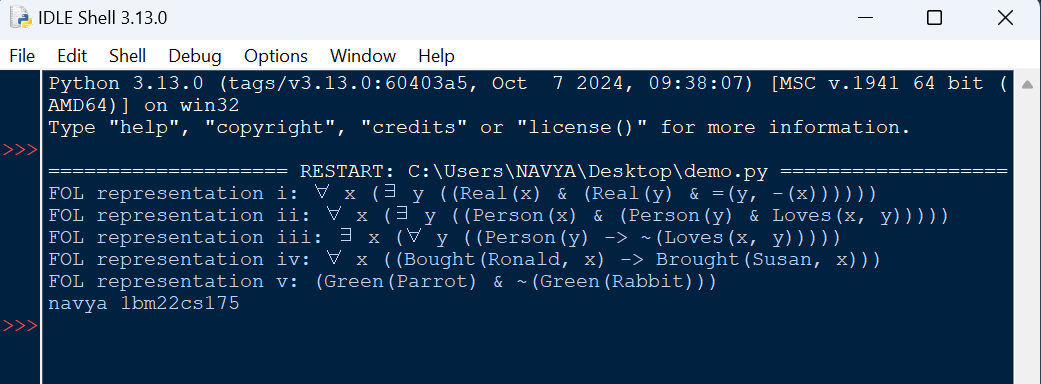
Green\_Rabbit = Predicate('Green', 'Rabbit')

expression\_v = Conjunction(Green, Negation(Green\_Rabbit))

print("FOL representation v:", expression\_v)

print("navya 1bm22cs175")

Output:

****

4a)

Code:

# 4a: Facts

facts = {

"Food": {"Apples", "Chicken", "Peanuts"}, # Initial known food items

"Eats": {"Bill": {"Peanuts"}}, # Bill eats peanuts

"Alive": {"Bill": True}, # Bill is alive

}

# Rules

def john\_likes\_food(x):

"""John likes all food."""

return x in facts["Food"]

def food\_from\_eating(y, x):

"""Anything anyone eats and isn't killed by is food."""

return x in facts["Eats"].get(y, set()) and facts["Alive"].get(y, False)

# Function to perform forward chaining

def forward\_chaining():

# Start with the known facts about food

inferred\_facts = set(facts["Food"])

# Step 1: Apply "Anything anyone eats and isn't killed by is food"

for person in facts["Eats"]:

for food in facts["Eats"][person]:

if food\_from\_eating(person, food): # If food is safe to eat

inferred\_facts.add(food) # Add it to food

# Step 2: Apply "John likes food" to all food items

for food in list(inferred\_facts): # We convert to list to avoid modifying while iterating

if john\_likes\_food(food):

inferred\_facts.add(f"Likes\_John\_{food}") # Add the fact that John likes the food

# Check if John likes peanuts

return "Likes\_John\_Peanuts" in inferred\_facts

# Add Peanuts as a food item if Bill eats peanuts and survives

facts["Eats"]["Bill"].add("Peanuts")

facts["Alive"]["Bill"] = True

# 1. Forward chaining to prove "John likes Peanuts"

print("Proving 'John likes peanuts' using forward chaining...")

result\_forward = forward\_chaining()

print("Result (Forward Chaining):", result\_forward) # Expected output: True

# Function to perform backward chaining

def backward\_chaining(goal):

# The goal is "Likes\_John\_Peanuts"

if goal == "Likes\_John\_Peanuts":

# To prove John likes peanuts, we need to show that Peanuts are food

if "Peanuts" in facts["Food"]:

return True

else:

# Check if Peanuts can be derived as food using the "Food\_from\_eating" rule

if food\_from\_eating("Bill", "Peanuts"):

facts["Food"].add("Peanuts") # Add Peanuts to the food set

return True

return False

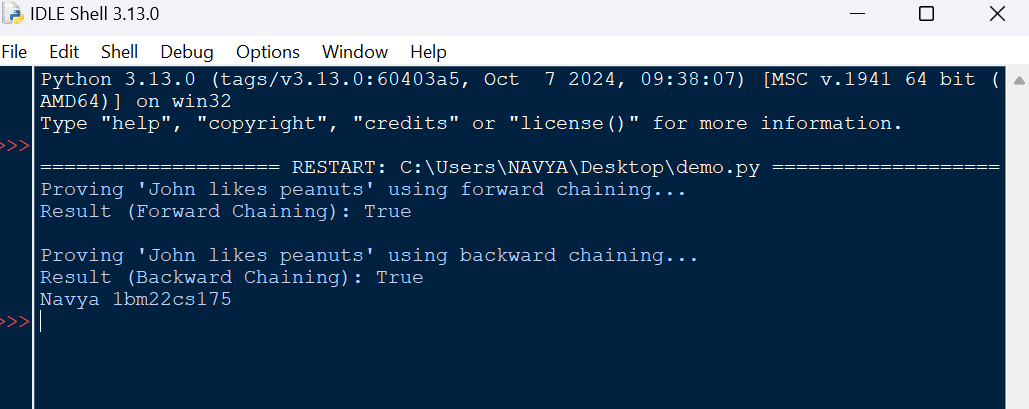
print("\nProving 'John likes peanuts' using backward chaining...")

result\_backward = backward\_chaining("Likes\_John\_Peanuts")

print("Result (Backward Chaining):", result\_backward) # Expected output: True

print("Navya 1bm22cs175")

Output:



4b)

Code:

# 4b: Minimax with Alpha-Beta Pruning

def minimax(node, depth, is\_maximizing\_player, values, alpha=float('-inf'), beta=float('inf')):

# Base case: If we reach a leaf node or exceed the depth

if depth == 0 or 2 \* node + 1 >= len(values):

return values[node] if node < len(values) else 0 # Return leaf node value or 0 if out of bounds

# If this is a MAX node

if is\_maximizing\_player:

best = float('-inf')

for i in range(2): # Two child nodes

child\_index = 2 \* node + 1 + i # Left and Right children

if child\_index < len(values): # Ensure child\_index is within bounds

child\_value = minimax(child\_index, depth - 1, False, values, alpha, beta)

best = max(best, child\_value)

alpha = max(alpha, best)

if beta <= alpha:

break # Beta cut-off

return best

# If this is a MIN node

else:

best = float('inf')

for i in range(2): # Two child nodes

child\_index = 2 \* node + 1 + i # Left and Right children

if child\_index < len(values): # Ensure child\_index is within bounds

child\_value = minimax(child\_index, depth - 1, True, values, alpha, beta)

best = min(best, child\_value)

beta = min(beta, best)

if beta <= alpha:

break # Alpha cut-off

return best

# Function to call minimax and simulate the game tree

def solve\_game\_tree():

# Leaf node values (given in the game tree)

values = [8, 9, 11, 10, 13, 12, 4, 6, 9, 6, 12, 14, 20, 2, 2, 2]

depth = 4 # Depth of the tree

root\_node = 0 # Start from the root node

# Start the minimax algorithm

result = minimax(root\_node, depth, True, values)

print(f"The optimal value for the root node is: {result}")

# Run the solution

solve\_game\_tree()

print("Navya 1bm22cs175")

Output:

