CIE-2

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")
```

```python
# Now, check if Mary sneezes
if sneeze("Mary"):
    print("Mary sneezes: True")
else:
    print("Mary sneezes: False")
print("Nikhilesh 1bm22cs181")
```

Output:

3B) code:

```python
# 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("Nikhilesh 1bm22cs181")
```

Output:

```
FOL representation i: ∀ x (∃ y ((Real(x) & (Real(y) & =(y, -(x))))))
FOL representation ii: ∀ x (∃ y ((Person(x) & (Person(y) & Loves(x, y)))))
FOL representation iii: ∃ x (∀ y ((Person(y) -> ~(Loves(x, y)))))
FOL representation iv: ∀ x ((Bought(Ronald, x) -> Brought(Susan, x)))
FOL representation v: (Green(Parrot) & ~(Green(Rabbit)))
Nikhilesh 1bm22cs181
>>> |
```

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("Nikhilesh 1bm22cs181")
```

output:

```
Proving 'John likes peanuts' using forward chaining...
Result (Forward Chaining): True

Proving 'John likes peanuts' using backward chaining...
Result (Backward Chaining): True
Nikhilesh 1bm22cs181
>>>
```

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("Nikhilesh 1bm22cs181")
```

Output:

```
The optimal value for the root node is: 9
Nikhilesh 1bm22cs181
>>>
```