LAB-2
ARTIFICIAL INTERLIGENCE
EC 9640

LAKSHAN W.G.

2020/E/079

GROUP EG10

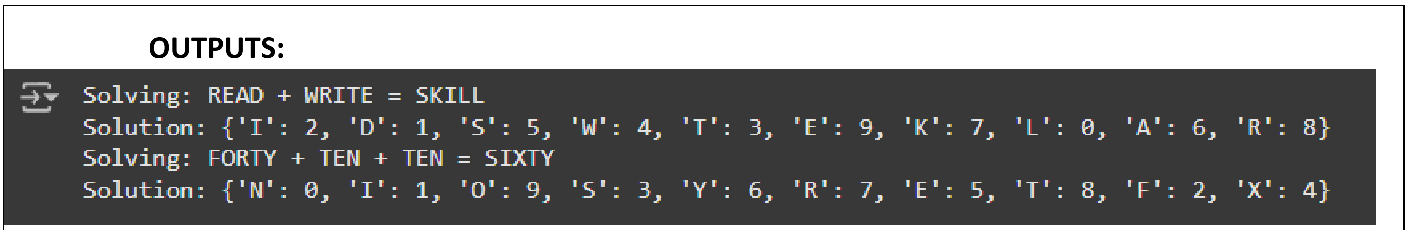
SEMESTER 7

02 JAN 2025

- **TASK 01:**

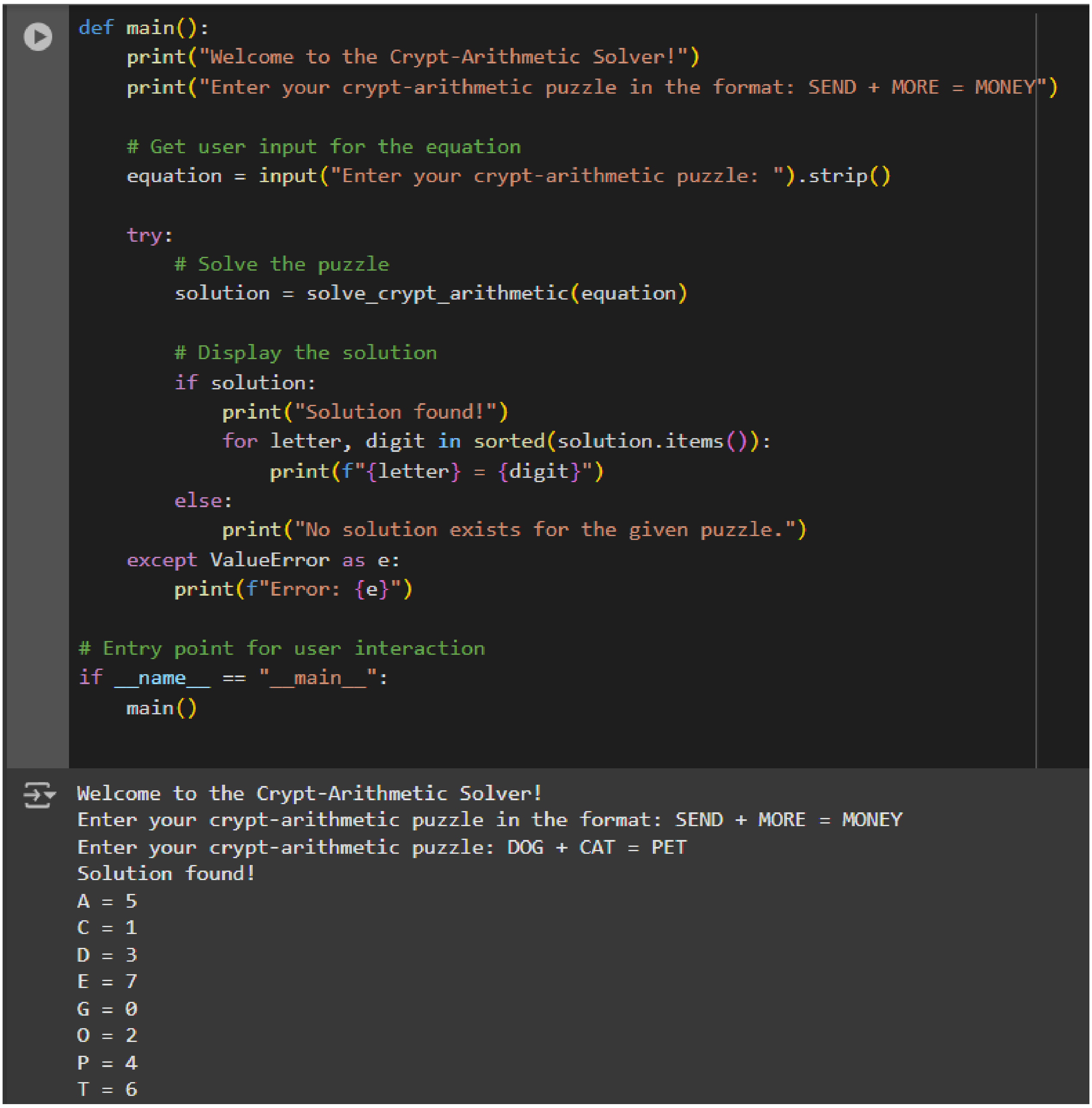
```
from itertools import permutations

def solve_crypt_arithmetic(equation):
    """
    Solve a crypt-arithmetic puzzle.

    :param equation: A string representing the puzzle (e.g., "SEND + MORE =
MONEY").
    :return: Solution as a dictionary or None if no solution exists.
    """
    # Remove spaces and split the equation
    equation = equation.replace(" ", "")
    left, right = equation.split("=")
    left_parts = left.split("+")

    # Extract unique letters
    unique_letters = set("".join(left_parts) + right)
    if len(unique_letters) > 10:
        raise ValueError("Too many unique letters. Maximum is 10.")

    # Generate permutations of digits
    for perm in permutations(range(10), len(unique_letters)):
        # Map letters to digits
        mapping = dict(zip(unique_letters, perm))

        # Ensure no leading digit is zero
        if any(mapping[word[0]] == 0 for word in [*left_parts, right]):
            continue

        # Substitute letters with digits in the equation
        left_nums = [int("".join(str(mapping[char]) for char in word)) for word in
left_parts]
        right_num = int("".join(str(mapping[char]) for char in right))

        # Check if the equation holds true
        if sum(left_nums) == right_num:
            return mapping

    return None

# Test the function with examples
examples = [
    "READ + WRITE = SKILL",
    "FORTY + TEN + TEN = SIXTY",
]

for example in examples:
    print(f"Solving: {example}")
    solution = solve_crypt_arithmetic(example)
    if solution:
        print("Solution:", solution)
    else:
        print("No solution found.")
```

OUTPUTS:



```
Solving: READ + WRITE = SKILL
Solution: {'I': 2, 'D': 1, 'S': 5, 'W': 4, 'T': 3, 'E': 9, 'K': 7, 'L': 0, 'A': 6, 'R': 8}
Solving: FORTY + TEN + TEN = SIXTY
Solution: {'N': 0, 'I': 1, 'O': 9, 'S': 3, 'Y': 6, 'R': 7, 'E': 5, 'T': 8, 'F': 2, 'X': 4}
```

More outputs using user inputs:



```python
def main():
    print("Welcome to the Crypt-Arithmetic Solver!")
    print("Enter your crypt-arithmetic puzzle in the format: SEND + MORE = MONEY")

    # Get user input for the equation
    equation = input("Enter your crypt-arithmetic puzzle: ").strip()

    try:
        # Solve the puzzle
        solution = solve_crypt_arithmetic(equation)

        # Display the solution
        if solution:
            print("Solution found!")
            for letter, digit in sorted(solution.items()):
                print(f"{letter} = {digit}")
        else:
            print("No solution exists for the given puzzle.")
    except ValueError as e:
        print(f"Error: {e}")

# Entry point for user interaction
if __name__ == "__main__":
    main()
```

```
Welcome to the Crypt-Arithmetic Solver!
Enter your crypt-arithmetic puzzle in the format: SEND + MORE = MONEY
Enter your crypt-arithmetic puzzle: DOG + CAT = PET
Solution found!
A = 5
C = 1
D = 3
E = 7
G = 0
O = 2
P = 4
T = 6
```

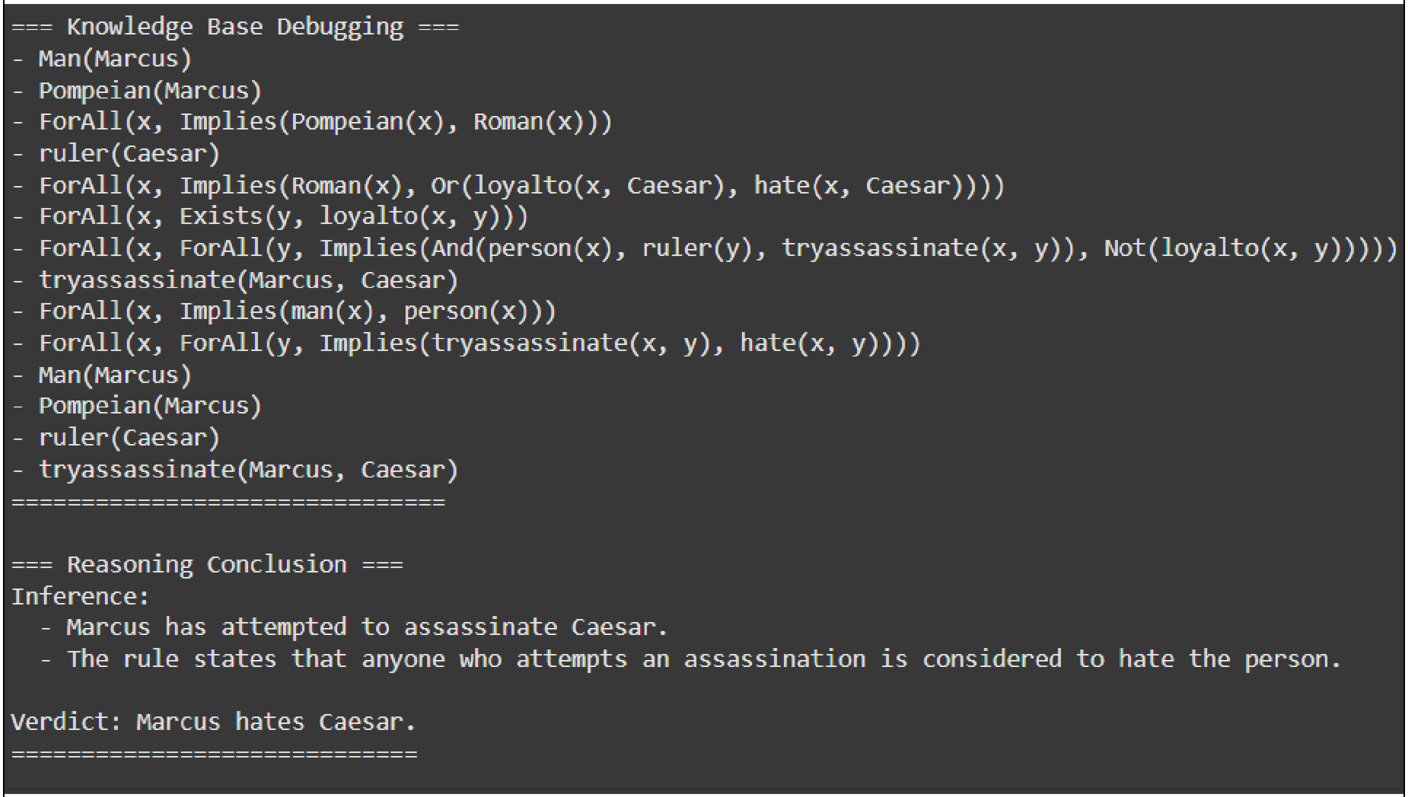
Key steps included:

- Parsing the input equation.
- Simplification of the equation by outlining the special letters of the alphabet.
- Making sure none of the digit in a particular place value is 0.
- generate all possible digit combinations for the unique letters.
- Verified the combinations by plugging in the digits into the equation to see if the arithmetic equation is correct.
- Printed the solution if there is a valid mapping; otherwise, printed No solution.

- **TASK 02:**

```
# Import libraries
import sys
import os

# Update path to where the 'aima' module is located in your Google Drive
sys.path.append('/content/drive/MyDrive/AI LAB 2/aima')
import aima.utils
import aima.logic

# Define the given facts and rules using logical expressions in FOL
logical_clauses = [
    aima.utils.expr("Man(Marcus)"),
    aima.utils.expr("Pompeian(Marcus)"),
    aima.utils.expr("ForAll(x, Implies(Pompeian(x), Roman(x)))"),  # If Pompeian,
then Roman
    aima.utils.expr("ruler(Caesar)"),
    aima.utils.expr("ForAll(x, Implies(Roman(x), Or(loyalto(x, Caesar), hate(x,
Caesar))))"),  # Roman leads to loyal or hate
    aima.utils.expr("ForAll(x, Exists(y, loyalto(x, y)))"),  # Existential rule for
loyalty
    aima.utils.expr("ForAll(x, ForAll(y, Implies(And(person(x), ruler(y),
tryassassinate(x, y)), Not(loyalto(x, y)))))"),  # Assassinate → no loyalty
    aima.utils.expr("tryassassinate(Marcus, Caesar)"),
    aima.utils.expr("ForAll(x, Implies(man(x), person(x)))"),  # man → person
    aima.utils.expr("ForAll(x, ForAll(y, Implies(tryassassinate(x, y), hate(x,
y))))")  # Assassination implies hate
]

# Initialize the knowledge base (KB) with the logical clauses
kb_instance = aima.logic.FolKB(logical_clauses)

# Add the facts to the knowledge base
kb_instance.tell(aima.utils.expr("Man(Marcus)"))
kb_instance.tell(aima.utils.expr("Pompeian(Marcus)"))
kb_instance.tell(aima.utils.expr("ruler(Caesar)"))
kb_instance.tell(aima.utils.expr("tryassassinate(Marcus, Caesar)"))

# Debugging: Output the current clauses in the knowledge base
print("=== Knowledge Base Debugging ===")
for clause in kb_instance.clauses:
    print(f"- {clause}")
print("================================")

# Perform logical reasoning to check if Marcus hates Caesar
result_of_hate_check = aima.logic.fol_fc_ask(kb_instance,
aima.utils.expr("hate(Marcus, Caesar)"))

# Output the result of the reasoning process
print("\n=== Reasoning Conclusion ===")
if result_of_hate_check:
    print("Inference:")
    print("  - Marcus has attempted to assassinate Caesar.")
    print("  - The rule states that anyone who attempts an assassination is considered to hate the person.")
    print("\nVerdict: Marcus hates Caesar.")
else:
    print("Inference:")
    print("  - The conditions to infer hatred from Marcus's actions and relationships are not met.")
    print("\nVerdict: Marcus does not hate Caesar.")
print("=============================")
```

OUTPUTS:

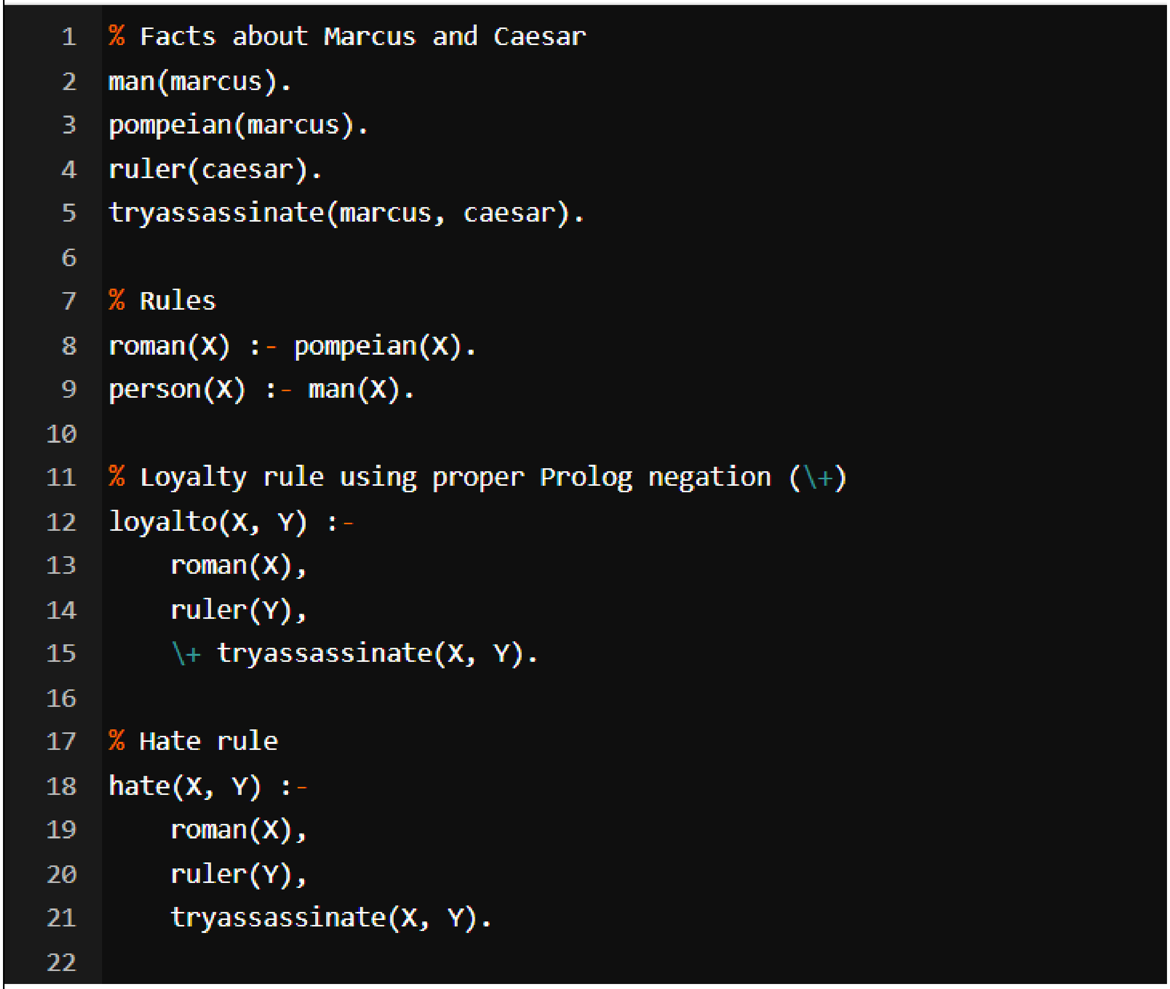


```
=== Knowledge Base Debugging ===
- Man(Marcus)
- Pompeian(Marcus)
- ForAll(x, Implies(Pompeian(x), Roman(x)))
- ruler(Caesar)
- ForAll(x, Implies(Roman(x), Or(loyalto(x, Caesar), hate(x, Caesar))))
- ForAll(x, Exists(y, loyalto(x, y)))
- ForAll(x, ForAll(y, Implies(And(person(x), ruler(y), tryassassinate(x, y)), Not(loyalto(x, y)))))
- tryassassinate(Marcus, Caesar)
- ForAll(x, Implies(man(x), person(x)))
- ForAll(x, ForAll(y, Implies(tryassassinate(x, y), hate(x, y))))
- Man(Marcus)
- Pompeian(Marcus)
- ruler(Caesar)
- tryassassinate(Marcus, Caesar)
===============================

=== Reasoning Conclusion ===
Inference:
  - Marcus has attempted to assassinate Caesar.
  - The rule states that anyone who attempts an assassination is considered to hate the person.

Verdict: Marcus hates Caesar.
=============================
```

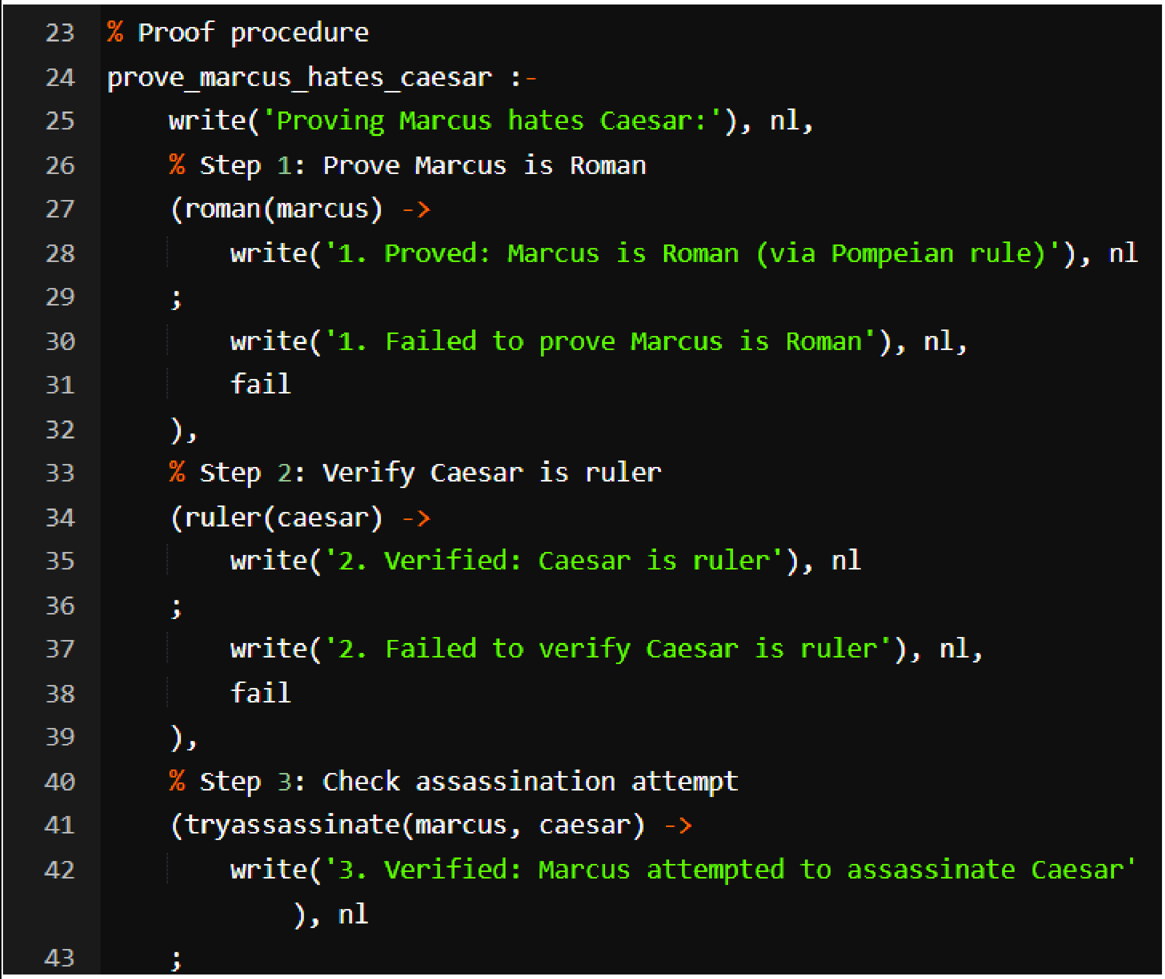
Key steps included:

- Encode first-order language at least with facts and rules.
- Logical expressions also have to be used to add facts and rules to the KB.
- Use forward chaining to draw new conclusions which are added into the KB.
- Determine if hate is true with relation to Marcus and Caesar.
- Reason, and thereby reach the conclusion that indeed Marcus must hate Caesar.

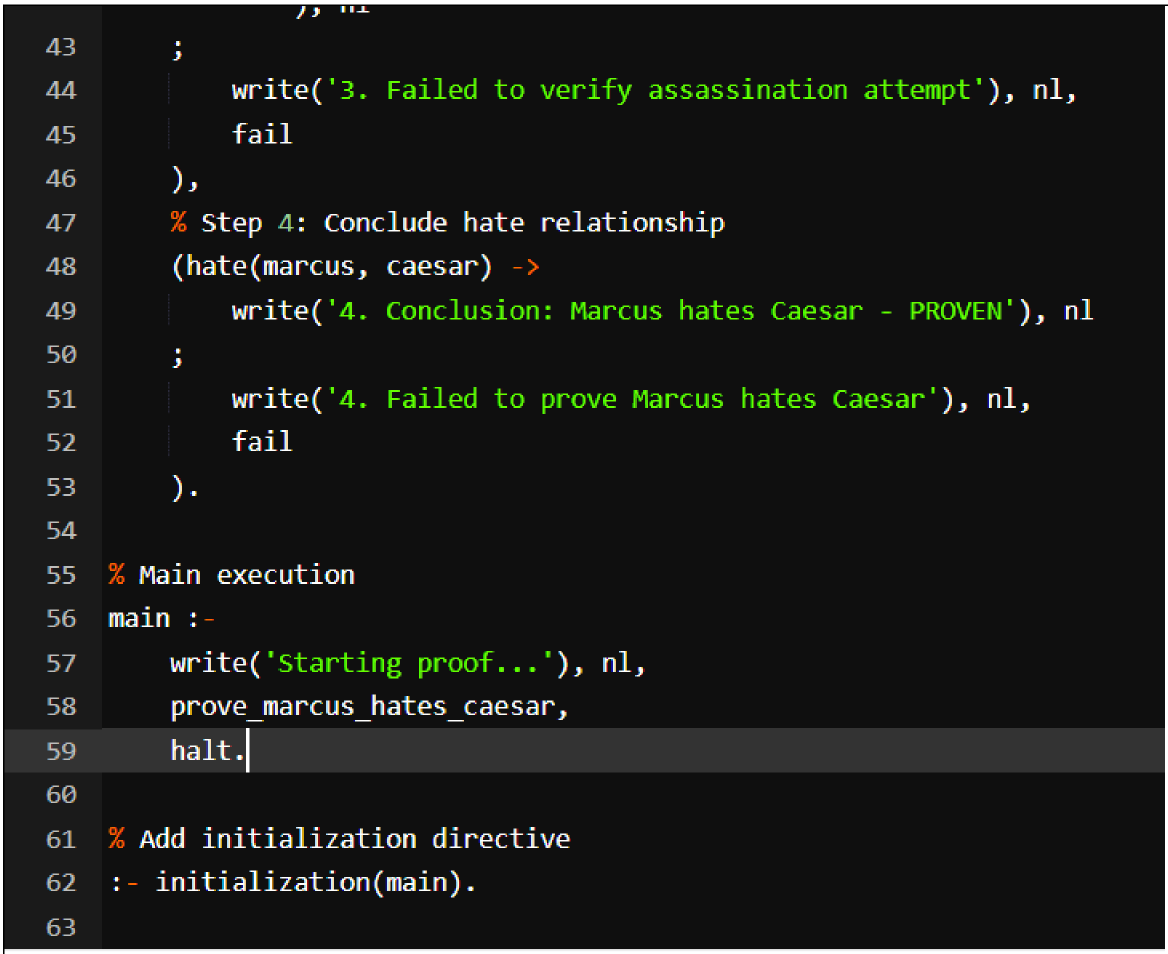
- **TASK 03: (used online prolog compiler)**



```prolog
% Facts about Marcus and Caesar
man(marcus).
pompeian(marcus).
ruler(caesar).
tryassassinate(marcus, caesar).

% Rules
roman(X) :- pompeian(X).
person(X) :- man(X).

% Loyalty rule using proper Prolog negation (\+)
loyalto(X, Y) :-
    roman(X),
    ruler(Y),
    \+ tryassassinate(X, Y).

% Hate rule
hate(X, Y) :-
    roman(X),
    ruler(Y),
    tryassassinate(X, Y).

```

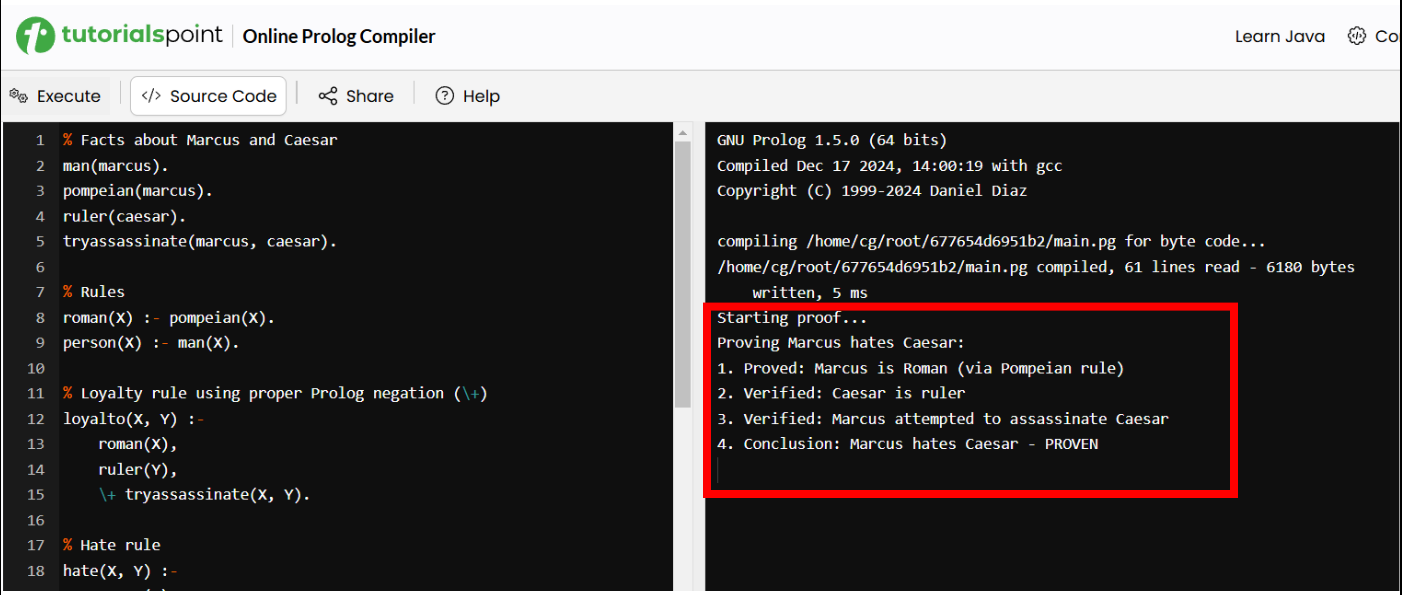


```prolog
% Proof procedure
prove_marcus_hates_caesar :-
    write('Proving Marcus hates Caesar:'), nl,
    % Step 1: Prove Marcus is Roman
    (roman(marcus) ->
        write('1. Proved: Marcus is Roman (via Pompeian rule)'), nl
    ;
        write('1. Failed to prove Marcus is Roman'), nl,
        fail
    ),
    % Step 2: Verify Caesar is ruler
    (ruler(caesar) ->
        write('2. Verified: Caesar is ruler'), nl
    ;
        write('2. Failed to verify Caesar is ruler'), nl,
        fail
    ),
    % Step 3: Check assassination attempt
    (tryassassinate(marcus, caesar) ->
        write('3. Verified: Marcus attempted to assassinate Caesar'), nl
    ;
```



```
            ), nl
    ;
        write('3. Failed to verify assassination attempt'), nl,
        fail
    ),
    % Step 4: Conclude hate relationship
    (hate(marcus, caesar) ->
        write('4. Conclusion: Marcus hates Caesar - PROVEN'), nl
    ;
        write('4. Failed to prove Marcus hates Caesar'), nl,
        fail
    ).

% Main execution
main :-
    write('Starting proof...'), nl,
    prove_marcus_hates_caesar,
    halt.

% Add initialization directive
:- initialization(main).
```

OUTPUT:



```
% Facts about Marcus and Caesar
man(marcus).
pompeian(marcus).
ruler(caesar).
tryassassinate(marcus, caesar).

% Rules
roman(X) :- pompeian(X).
person(X) :- man(X).

% Loyalty rule using proper Prolog negation (\+)
loyalto(X, Y) :-
    roman(X),
    ruler(Y),
    \+ tryassassinate(X, Y).

% Hate rule
hate(X, Y) :-
```

```
GNU Prolog 1.5.0 (64 bits)
Compiled Dec 17 2024, 14:00:19 with gcc
Copyright (C) 1999-2024 Daniel Diaz

compiling /home/cg/root/677654d6951b2/main.pg for byte code...
/home/cg/root/677654d6951b2/main.pg compiled, 61 lines read - 6180 bytes
    written, 5 ms
Starting proof...
Proving Marcus hates Caesar:
1. Proved: Marcus is Roman (via Pompeian rule)
2. Verified: Caesar is ruler
3. Verified: Marcus attempted to assassinate Caesar
4. Conclusion: Marcus hates Caesar - PROVEN
```

Key steps included:

- This was done using the rule of Pompeian which states that if an individual is a Pompeian, then their nationality is Roman (as Marcus is a Pompeian he is also Roman).
- Confirmed that Caesar is a ruler by checking on the verified fact from the computer.
- Ensured that supported the fact, which stated that Marcus tried to kill Caesar. Conclusion: In the light of hate rule, it was ascertained that Marcus hated Caesar.