Assignment 4 - Question 6

NAME: Patel, Aditya Kamleshkumar
NAME: Osadebe, Osanyem

LLM Used for Testing

We used ChatGPT 4o, specifically for constraint satisfaction and Prolog queries. We wanted to test this LLM on the Prolog cryptarithmetic problem in Assignment 3, specifically, the pure generate and test approach.

Prompting and Adjustments

Initial Prompt

Our first prompt asked the model to solve a problem with prolog constraint satisfaction by pretending to be an expert in prolog and CSPs and we provided an example of a solve map colouring CSP. The LLM gave a basic constraint structure but missed the arithmetic constraints and digit assignments for a cryptarithmetic puzzle. The code had syntax errors. This showed that we needed to be more specific and reinforce the digit constraints in the prompt.

Second Prompt

We adjusted the prompt to ask each letter to be a unique digit with no leading zeroes. This helped. The LLM gave variable definitions and initial arithmetic constraints and was closer to a solution. However, it still gave multiple outputs and did not produce the one correct solution. Logical errors were still present because of the lack of distinctness constraints across variables, which resulted in repeated outputs. This showed the model needs more guidance on distinct digit rules and Prolog-specific predicates for uniqueness.

Final Prompt

In the final prompt, we asked JET * AX = LOVE, specifying a standard multiplication example, distinct digit constraints, and arithmetic verification. This made the LLM give a response that conforms to Prolog syntax and uses the right uniqueness predicates. The output was more accurate, and enforced distinct digit constraints across all variables.

Conclusion

Through prompting the LLM, we got better and better at addressing the specific requirements for syntax, logical constraints, and prolog predicates. Initially, it had syntax errors, missing distinctness rules, and giving incorrect answers. The LLM gave a working solution at the end by refining the prompt to specify unique digit constraints, leading to zero restrictions and arithmetic verification. However, manual verification and adjustments were needed because the model sometimes got Prolog syntax and constraint satisfaction principles wrong.

Input (Final Prompt):
Imagine yourself as an expert in Prolog and your are also an expert in solving Constraint Satisfaction Problems (CSPs).

Here is an example of a constraint satisfaction problem - a map coloring problem:
You are tasked with coloring a map consisting of four regions: a, b, c, and d. The regions are adjacent to each other as follows:

Region a is adjacent to regions b
Region a is adjacent to regions d
Region c is adjacent to regions b
Region c is adjacent to regions d
You need to assign one of three colors (red, green, or blue) to each region, ensuring that no two adjacent regions share the same color. Your task is to write a Prolog program that solves this map coloring problem.

NB: Adjacency is always transitive, i.e., if a is adjacent to b, then b is adjacent to a

Requirements:
Define the domain of possible colors.
Write a solve/1 predicate that finds a valid coloring for all four regions.
Ensure that adjacent regions do not share the same color.

And here is how it should be solved:
% Domain - Possible Colors
color(red).
color(green).
color(blue).

% Main predicate to solve the map coloring problem

```
solve([A, B, C, D]) :-
    color(A), color(B), color(C), color(D),

    % No adjacent regions can share the same color
    not A = B, % a and b must have different colors
    not A = D, % a and d must have different colors
    not B = C, % b and c must have different colors
    not C = D. % c and d must have different colors
```

Using this knowledge and your expertise, solve this cryptarithmetic problem in prolog.

```
          J E T
*            A X
------------------
+         A X L E
        J E T .
-----------------
        L O V E
```

using the following constraints:
Each letter stands for a distinct digit and leading digits are not zeroes. You cannot make any other assumptions.

Try to solve this problem using the pure generate and test technique, without any in-
terleaving, and notice how much time it takes. In particular, you should write the rules for solve,
which takes in your list of variables and finds an assignment that solves the puzzle.

This notation means that JET * AX = LOVE, and the words between the bars correspond to performing the usual multiplication algorithm. That is AXLE + 10*JET = LOVE. For example, the calculation of 123 * 12 would appear as:

```
          1 2 3
*            1 2
------------------
+           2 4 6
        1 2 3 .
-----------------
        1 4 7 6
```

Add comments to the code to explain your reasoning

PROLOG Instructions: When you write your rules in PROLOG, you are not allowed to use “;” (disjunction), “!” (cut),
and “->” (if-then). You are only allowed to use “;” to get additional responses when interacting with PROLOG from
the command line. Note that this is equivalent to using the “More” button in the ECLiPSe GUI. We will be using ECLiPSE Prolog release 6.

ChatGPT’s Final Response:

To solve this cryptarithmetic puzzle in Prolog using the generate-and-test technique, let's start by defining the problem's constraints step-by-step.

Problem Breakdown

1. **JET * AX = LOVE:** We need to find digits such that the product of the three-digit number JETJETJET and the two-digit number AXAXAX results in the four-digit number LOVELOVELOVE.
2. **AXLE + 10 * JET = LOVE:** According to the multiplication algorithm, we also need to satisfy this secondary condition, which represents how the multiplication aligns in the standard column multiplication format.
3. **Unique Digits Constraint:** Each letter represents a unique digit, and leading digits cannot be zero.

Steps to Solve the Problem

1. **Define Domain:** Each letter can represent a digit from 0 to 9, but `J`, `A`, and `L` (the leading digits of JETJETJET, AXAXAX, and LOVELOVELOVE) cannot be zero.
2. **Generate Variables:** We define variables for each letter (`J`, `E`, `T`, `A`, `X`, `L`, `O`, `V`, `E`).
3. **Define Constraints:** Using Prolog constraints, we ensure that:
   - `JET` and `AX` form valid three- and two-digit numbers respectively.
   - `AXLE` + 10 * `JET` = `LOVE`.
4. **Uniqueness of Digits:** All letters must represent distinct digits.

Here's the Prolog code that follows this approach.

Prolog Code

```
% Define the digit domain for each variable from 0 to 9.
% Leading digits will have additional constraints (non-zero).
```

```
digit(0). digit(1). digit(2). digit(3). digit(4).
digit(5). digit(6). digit(7). digit(8). digit(9).

% The main predicate solve/1 attempts to find a valid solution for the variables [J, E, T, A, X, L, O, V].
solve([J, E, T, A, X, L, O, V]) :-

    % Assign each variable a value within the digit domain, with additional constraints on leading digits.
    % J, A, and L cannot be zero, as they are leading digits of multi-digit numbers (JET, AX, and LOVE
respectively).
    digit(J), J \= 0,     % J cannot be 0 (leading digit of JET).
    digit(E),
    digit(T),
    digit(A), A \= 0,     % A cannot be 0 (leading digit of AX).
    digit(X),
    digit(L), L \= 0,     % L cannot be 0 (leading digit of LOVE).
    digit(O),
    digit(V),

    % Ensure all variables (digits) are unique, as each letter represents a distinct digit.
    all_different([J, E, T, A, X, L, O, V]),

    % Calculate the numeric values of JET, AX, AXLE, and LOVE from the individual digit assignments.
    % These variables represent the multi-digit numbers in the problem.
    JET is 100*J + 10*E + T,           % JET is a 3-digit number: 100*J + 10*E + T.
    AX is 10*A + X,                    % AX is a 2-digit number: 10*A + X.
    AXLE is 1000*A + 100*X + 10*L + E, % AXLE is a 4-digit number: 1000*A + 100*X + 10*L + E.
    LOVE is 1000*L + 100*O + 10*V + E, % LOVE is a 4-digit number: 1000*L + 100*O + 10*V + E.

    % Enforce the main cryptarithmetic constraint from the puzzle.
    % This constraint represents the multiplication formula: JET * AX must equal LOVE.
    JET * AX =:= LOVE,

    % Secondary constraint derived from the column-by-column multiplication.
    % AXLE + (10 * JET) must equal LOVE for the solution to be valid.
    AXLE + 10*JET =:= LOVE.

% Helper predicate to enforce that all elements in a list are distinct.
% This predicate recursively checks each element in the list and ensures it is not repeated.
all_different([]).                    % Base case: an empty list has all unique elements.
all_different([H|T]) :-               % Recursive case: take the head element H and tail list T.
    \+ member(H, T),                  % Ensure H is not in T (i.e., H is unique in the list).
    all_different(T).                 % Recursively check the tail list for uniqueness.
```

Explanation of Code

1. **Defining Digits:** `digit/1` represents possible values for each letter.
2. **Solve Predicate:** `solve/1` takes a list of variables `[J, E, T, A, X, L, O, V]` and assigns values to them that satisfy the constraints.
3. **JET, AX, AXLE, and LOVE Calculations:** We calculate `JET`, `AX`, `AXLE`, and `LOVE` using the assigned digits, treating them as multi-digit numbers.
4. **Constraints:** `JET * AX =:= LOVE` and `AXLE + 10 * JET =:= LOVE` enforce the cryptarithmetic conditions.
5. **Unique Digits Constraint:** `all_different/1` ensures all assigned digits are distinct.

Testing the Program

Once the code is written, you can test it in ECLiPSe Prolog by running:

```
?- solve([J, E, T, A, X, L, O, V]).
```

This code uses the pure generate-and-test approach without cuts or disjunctions. ECLiPSe Prolog will search through all possible assignments, which may take time due to the number of constraints and the generate-and-test method's inherent inefficiency in a combinatorial search.