**Queens College of CUNY**
**Department of Computer Science**
**Programming Languages**
**(CSCI 316)**
**Winter 2026**

# Assignment #12
# "Logic Programming Languages"
**Due: January 22, 2026**

## Introduction:

"Logic Programming" is a programming, database and knowledge representation paradigm based on formal logic. A logic program is a set of sentences in logical form, representing knowledge about some problem domain. Computation is performed by applying logical reasoning to that knowledge, to solve problems in the domain. Rules are written in the form of clauses and are read as declarative sentences in logical form:

In this assignment, we solve some classic problems using logic programming.

## Submissions:

In the Google form, please submit:

- Assignmen12.pl (source code)
- Assignment12.txt (console output)

## Tasks:

Review and the code for these four examples, and collect output:

- Family Tree
- Sieve of Eratosthenes
- Latin Square (like Sudoku but without sub-block constraints)
- n-Queens

### Facts

```
parent(abraham, isaac).
parent(isaac, jacob).
parent(jacob, joseph).
parent(jacob, judah).
```

Each fact states something that is unconditionally true.

### Rules

```
grandparent(X, Z) :-
    parent(X, Y),
    parent(Y, Z).
```

This rule says:

X is a grandparent of Z if X is a parent of Y and Y is a parent of Z.

### Sample queries

```
?- parent(jacob, joseph).
true.

?- grandparent(abraham, jacob).
true.

?- grandparent(X, joseph).
X = isaac.

?- parent(jacob, X).
X = joseph ;
X = judah.
```

```
% ============================================================
% Sieve of Eratosthenes in Prolog
% ------------------------------------------------------------
% PURPOSE:
%   Find all prime numbers less than or equal to a given N.
%
% DEFINITION:
%   A prime number is an integer greater than 1 that has
%   no positive divisors other than 1 and itself.
%
% METHOD:
%   This program implements the classical Sieve of Eratosthenes:
%     1) Start with the list [2, 3, 4, ..., N]
%     2) Take the first number P in the list (it is prime)
%     3) Remove all multiples of P from the rest of the list
%     4) Repeat with the remaining numbers
%
% This is written in a clear, instructional Prolog style.
% ============================================================

% ------------------------------------------------------------
% sieve(+N, -Primes)
%
% N       : upper bound (must be >= 2)
% Primes  : list of all prime numbers <= N
%
% Example:
%   ?- sieve(30, P).
%   P = [2,3,5,7,11,13,17,19,23,29].
% ------------------------------------------------------------

sieve(N, Primes) :-
    N >= 2,

    % Create the initial list of candidate numbers:
    % [2, 3, 4, ..., N]
    numlist(2, N, Numbers),

    % Apply the sieve process to the list
    sieve_list(Numbers, Primes).

% ------------------------------------------------------------
% sieve_list(+Candidates, -Primes)
%
% Candidates : numbers still under consideration
% Primes     : primes extracted from Candidates
%
% Algorithm:
%   - If the candidate list is empty, we are done
%   - Otherwise:
%       * Take the head H (a prime)
%       * Remove all multiples of H from the tail
%       * Continue sieving the remaining list
% ------------------------------------------------------------

sieve_list([], []).   % No candidates left → no primes left

sieve_list([H|T], [H|Primes]) :-
    % H is the smallest remaining number, therefore prime

    % Remove all numbers in T that are divisible by H
    remove_multiples(H, T, Filtered),

    % Continue sieving with the filtered list
    sieve_list(Filtered, Primes).

% ------------------------------------------------------------
% remove_multiples(+P, +List, -Result)
%
% P      : a prime number
% List   : list of candidate numbers
% Result : List with all multiples of P removed
%
% This predicate scans List recursively and keeps only those
% numbers that are NOT divisible by P.
% ------------------------------------------------------------

remove_multiples(_, [], []).
    % Base case: empty input list → empty result list

remove_multiples(P, [H|T], Result) :-
    (   H mod P =:= 0
        % If H is divisible by P, it is NOT prime
        % so we discard it and continue with the tail
    ->  remove_multiples(P, T, Result)
    ;   % Otherwise, H is kept in the result list
        Result = [H|Rest],
        remove_multiples(P, T, Rest)
    ).

% ------------------------------------------------------------
% HOW TO RUN
% ------------------------------------------------------------
% 1) Load the file:
%    ?- [sieve].
%
% 2) Ask for primes up to N:
%    ?- sieve(50, Primes).
%
% 3) Result:
%    Primes = [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47].
%
% ----------------------------------------------------------
% CONCEPTS DEMONSTRATED
% ----------------------------------------------------------
% - Declarative problem description
% - List recursion and pattern matching
% - Guards using arithmetic (mod, =:=)
% - Logical variables and backtracking
% - Classic algorithm expressed in Prolog style
% ==========================================================

% ==========================================================
% Latin Square Solver: 5 x 5 (symbols 1..5)
% ----------------------------------------------------------
% This is NOT Sudoku.
% It enforces only:
%   - Row uniqueness
%   - Column uniqueness
%
% Input format:
%   - A list of 5 rows
%   - Each row is a list of 5 integers
%   - Use 0 to represent a blank (unknown) cell
%
% Example puzzle:
%   P = [
%     [1,0,0,0,5],
%     [0,2,0,0,0],
%     [0,0,3,0,0],
%     [0,0,0,4,0],
%     [5,0,0,0,1]
%   ].
%
% Solve and print:
%   ?- solve5_print(P).
%
% Or keep the solution:
%   ?- solve5_puzzle(P, Solution).
% ==========================================================

:- use_module(library(clpfd)).   % Finite-domain constraints

% ----------------------------------------------------------
% solve5(+Grid)
% Grid is a 5x5 matrix (list of 5 lists, each length 5).
%
```

```
% This predicate:
%   1) checks the grid shape
%   2) constrains all cells to be in 1..5
%   3) enforces all_distinct on rows
%   4) enforces all_distinct on columns
%   5) labels variables to find a concrete solution
% -----------------------------------------------------------
solve5(Grid) :-
    % --- Must have exactly 5 rows
    length(Grid, 5),

    % --- Each row must have exactly 5 columns
    maplist(same_length([_,_,_,_,_]), Grid),

    % --- Flatten the grid to get all variables/cells
    append(Grid, Vars),

    % --- Domain constraint: values are 1 through 5
    Vars ins 1..5,

    % --- Row constraints: no duplicates in any row
    maplist(all_distinct, Grid),

    % --- Column constraints: transpose rows into columns
    transpose(Grid, Cols),
    maplist(all_distinct, Cols),

    % --- Search strategy to assign values
    labeling([ff], Vars).

% -----------------------------------------------------------
% puzzle5_to_vars(+Puzzle0, -Grid)
% Converts a puzzle with 0 = blank into a grid of FD variables.
% -----------------------------------------------------------
puzzle5_to_vars(Puzzle0, Grid) :-
    % Check the puzzle has correct 5x5 shape
    length(Puzzle0, 5),
    maplist(same_length([_,_,_,_,_]), Puzzle0),

    % Convert each row
    maplist(row5_to_vars, Puzzle0, Grid).

% -----------------------------------------------------------
% row5_to_vars(+Row0, -Row)
% Converts one row from integers/0s into integers/variables.
% -----------------------------------------------------------
row5_to_vars(Row0, Row) :-
    maplist(cell5_to_var, Row0, Row).

% -----------------------------------------------------------
% cell5_to_var(+CellIn, -CellOut)
%  - 0  -> blank, becomes a fresh Prolog variable
%  - 1..5 -> fixed value, constrained to domain
% -----------------------------------------------------------
```

```
cell5_to_var(0, _) :- !.
cell5_to_var(N, N) :-
   N in 1..5.

% -----------------------------------------------------------
% solve5_puzzle(+Puzzle0, -Solution)
% Full pipeline: convert puzzle, apply constraints, solve.
% -----------------------------------------------------------
solve5_puzzle(Puzzle0, Solution) :-
   puzzle5_to_vars(Puzzle0, Solution),
   solve5(Solution).

% -----------------------------------------------------------
% solve5_print(+Puzzle0)
% Convenience predicate: solve and print row by row.
% -----------------------------------------------------------
solve5_print(Puzzle0) :-
   solve5_puzzle(Puzzle0, Sol),
   maplist(writeln, Sol).

% -----------------------------------------------------------
% sample5(-Puzzle)
% A small example puzzle.
% -----------------------------------------------------------
sample5([
  [1,0,0,0,5],
  [0,2,0,0,0],
  [0,0,3,0,0],
  [0,0,0,4,0],
  [5,0,0,0,1]
]).

% -----------------------------------------------------------
% Demo:
%   ?- sample5(P), solve5_print(P).
% -----------------------------------------------------------
```

================================================================================

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  N-Queens Problem (8 Queens example)
%
%  Goal:
%    Place N queens on an N×N chessboard so that:
%      - exactly one queen is in each row
%      - exactly one queen is in each column
%      - no two queens attack each other diagonally
%
%  Representation:
%    A solution is a list Qs of length N.
%    The index of the list = row number
%    The value at each position = column number
%
```

```
%    Example:
%      Qs = [1,5,8,6,3,7,2,4]
%      means:
%        Row 1 → Column 1
%        Row 2 → Column 5
%        Row 3 → Column 8
%        ...
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Entry point for 8 queens
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% queens8(-Qs)
% Qs will be bound to a solution for the 8-queens problem
queens8(Qs) :-
    nqueens(8, Qs).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% General N-Queens solver
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% nqueens(+N, -Qs)
% N  = size of the board (N×N)
% Qs = list of length N representing a solution
nqueens(N, Qs) :-
    % Generate the list [1, 2, ..., N]
    % Each number represents a column
    numlist(1, N, Cols),

    % Generate a permutation of columns
    % This ensures:
    %   - one queen per row (one list element per row)
    %   - one queen per column (no duplicates)
    permutation(Cols, Qs),

    % Check that no queens attack each other diagonally
    safe(Qs).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Safety checks
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% safe(+Qs)
% True if all queens in Qs are mutually non-attacking
safe([]).
safe([Q | Qs]) :-
    % First ensure the rest of the queens are safe
    safe(Qs),

    % Then ensure Q does not attack any queen in Qs
    no_attack(Q, Qs, 1).
```

```
% no_attack(+Q, +Qs, +D)
% Q  = column of the current queen
% Qs = columns of queens in later rows
% D  = distance in rows between queens
no_attack(_, [], _).
no_attack(Q, [Q2 | Qs], D) :-
    % Different columns
    % (Technically guaranteed by permutation/2,
    %  but kept here for clarity)
    Q =\= Q2,

    % Different diagonals:
    % If the column difference equals the row difference,
    % the queens are on the same diagonal → forbidden
    abs(Q - Q2) =\= D,

    % Move to the next row distance
    D1 is D + 1,
    no_attack(Q, Qs, D1).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Optional: Pretty-print board
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% print_board(+Qs)
% Displays a visual representation of the board
print_board(Qs) :-
    length(Qs, N),
    forall(
        nth1(Row, Qs, Col),
        (
            % For each column in the row
            forall(
                between(1, N, C),
                (
                    % Print 'Q' where the queen is,
                    % '.' elsewhere
                    ( C =:= Col ->
                        write('Q ')
                    ;
                        write('. ')
                    )
                )
            ),
            nl
        )
    ).
```

**Save the code**

Save the code in a file, for example:queens.pl

**Start Prolog**

From a terminal or command prompt:swipl

**Load the file**

At the Prolog prompt:?- [queens] or ?- consult('queens.pl').

**Get one 8-queens solution**

?- queens8(Qs).

Example result:

Qs = [1, 5, 8, 6, 3, 7, 2, 4] ;

Press:

- ; to ask for another solution
- . to stop

**Print a board for a solution**

?- queens8(Qs), print_board(Qs).

```
Output looks like:

Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .
```

Press ; to see another board.

**Solve for a different board size**

Because the code is general:

?- nqueens(4, Qs).

?- nqueens(10, Qs).

**Count all solutions (optional)**

```
?- findall(Qs, queens8(Qs), Solutions),
   length(Solutions, Count).
```

Result:

Count = 92.