Declarative Programming Project 2 Spec (Copied from Grok)

A maths puzzle is a square grid of squares, each to be filled in with a single digit 1–9 (zero is not permitted) satisfying these constraints:

• each row and each column contains no repeated digits;

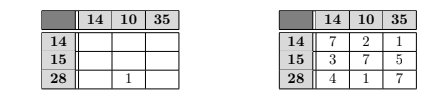
• all squares on the diagonal line from upper left to lower right contain the same value; and

• the heading of reach row and column (leftmost square in a row and topmost square in a column) holds either the sum or the product of all the digits in that row or column

Note that the row and column headings are not considered to be part of the row or column, and so may be filled with a number larger than a single digit. The upper left corner of the puzzle is not meaningful.

When the puzzle is originally posed, most or all of the squares will be empty, with the headings filled in. The goal of the puzzle is to fill in all the squares according to the rules. A proper maths puzzle will have at most one solution.

Here is an example puzzle as posed (left) and solved (right) :



Project 2

You will write Prolog code to solve maths puzzles. Your program should supply a predicate **puzzle\_solution(Puzzle)** that holds when Puzzle is the representation of a solved maths puzzle.

A maths puzzle will be represented as a list of lists, each of the same length, representing a single row of the puzzle. The first element of each list is considered to be the header for that row. Each element but the first of the first list in the puzzle is considered to be the header of the corresponding column of the puzzle. The first element of the first element of the list is the corner square of the puzzle, and thus is ignored.

You can assume that when your puzzle\_solution/1 predicate is called, its argument will be a proper list of proper lists, and all the header squares of the puzzle (plus the ignored corner square) are bound to integers. Some of the other squares in the puzzle may also be bound to integers, but the others will be unbound. When puzzle\_solution/1 succeeds, its argument must be ground. You may assume your code will only be tested with proper puzzles, which have at most one solution. Of course, if the puzzle is not solvable, the predicate should fail, and it should never succeed with a puzzle argument that is not a valid solution. For example,  
your program would solve the above puzzle as below:

?- Puzzle=[[0,14,10,35],[14,\_,\_,\_],[15,\_,\_,\_],[28,\_,1,\_]],  
|   puzzle\_solution(Puzzle).  
Puzzle = [[0, 14, 10, 35], [14, 7, 2, 1], [15, 3, 7, 5], [28, 4, 1, 7]] ;  
false.

Your puzzle\_solution/1 predicate, and all supporting code, should be written in the file **proj2.pl**. You may also use Prolog library modules supported by SWI Prolog as installed on the server, which is version 7.2.3. You may, but need not, use SWI Prolog’s Constraint Logic Programming facilities to solve the problem.

**Hints:**

1. Begin by unifying all the squares on the diagonal. This only needs to be done once to ensure that the puzzle satisfies the first constraint.

2. It is fairly simple to check if a row of the puzzle (one of the inner lists) is a valid solution. Simply check that the first list element is equal to the sum or product of the other elements. Checking the columns is a bit more complicated. However, the library provides a **transpose/2** predicate. If you transpose the puzzle, the columns become rows, so you can check the columns by checking the rows of the transposed puzzle. Load the library module containing the transpose/2 predicate with the directive

:- ensure loaded(library(clpfd)).

3. A very simple strategy for solving puzzles is to backtrack over all possible values for each puzzle square, and test that it is a valid puzzle solution. For a 2×2 puzzle, there are 94 = 6561 filled puzzles. This is very tractable, as long as the code to check that a puzzle is valid is reasonably efficient. This is a good way to get started.

4. However, even for a 3×3 puzzle, there are 99 = 387,420,489 filled puzzles, and even for very efficient code to check if a solution is valid, the strategy of Hint 3 will not be tractable. But note that if the first row of the filled puzzle is not valid, the whole puzzle will be invalid, so there is no point exploring those possibilities. For example, if the heading of the first row is 23, only 6 of the 93 = 729 possible ways to fill it could possibly be valid. More importantly, if you check that each row and column is valid as soon as it is generated, rather than generating the whole grid before checking, you can cut the search space enormously. If all three rows of a 3×3 puzzle have only 6 solutions, then there are only 63 = 216 candidate puzzles to check (since once all the rows have been filled in, then so have all the columns).

5. For some headings, there are far more than 6 possible rows or columns. For example, a three square row or column with a heading of 16 has 54 valid ways to fill it, and for a four square row or column, there are 192 possibilities. Thus for some puzzles, it will be necessary to further cut the number of possibilities to explore.

One way to do this is to first fill the row or column with the fewest possibilities. Consider the example puzzle: the last row has only two possible ways to fill it: 7—1—4 and 4— 1—7, while the last column has six possible ways.

Once one row or column is filled, that will reduce the number of possibilities in other rows and columns. In the example above, if the last row is filled with 7—1—4, There are, in fact, no ways to fill the last column, thus the choice of 7—1—4 for the last row can be immediately dismissed (since no selection for any of the other squares can get around the fact that there are no two digits that can be added to or multiplied by 4 to get 35). This leaves 4—1—7 as the only possible way to fill in the last row. By repeatedly choosing the row or column with the fewest alternatives and (non-deterministically) filling it with valid values, you can achieve adequate performance for this project.

6. The easiest way to do all this in Prolog is to reason about which variables are bound and which are unbound as a way to determine which squares have already been filled. For example, you can use the built-in predicate **ground(Term)**, which holds when Term is ground at the time the test is performed. When this succeeds for the puzzle, that means the entire puzzle has been filled in. At this point you can check that it is bound to a valid solution, and if so, succeed. If the puzzle is not ground, you will need to select a row or column and fill it in. Doing this will fill in one square in all perpendicular columns or rows, substantially cutting down on the number of valid ways they can be filled. In fact, if you have taken Hint 1, then filling in one row will fill in one square in all the other rows, and one or two squares in all of the columns, and the reverse if you fill a column.

The easiest way to decide which row or column has the fewest solutions is to use the **bagof/3** predicate, and take the length of the resulting list of solutions.

**Assessment**

Your project will be assessed on the following criteria:

**30%** Quality of your code and documentation;

**20%** The ability of your program to correctly solve 2×2 maths puzzles;

**30%** The ability of your program to correctly solve 3×3 maths puzzles; and

**20%** The ability of your program to correctly solve 4×4 maths puzzles.

Note that timeouts will be imposed on all tests. You will have at least 20 seconds to solve each puzzle, regardless of difficulty. Executions taking longer than that will be unceremoniously terminated, leading to that test being assessed as failing. Twenty seconds should be ample with careful implementation.

See the Project Coding Guidelines on the LMS for detailed suggestions for coding style. These guidelines will form the basis of the quality assessment of your code and documentation. Be sure to document the predicates you take from the starter file you are given, too.