Lab 1:
Puzzling Prolog

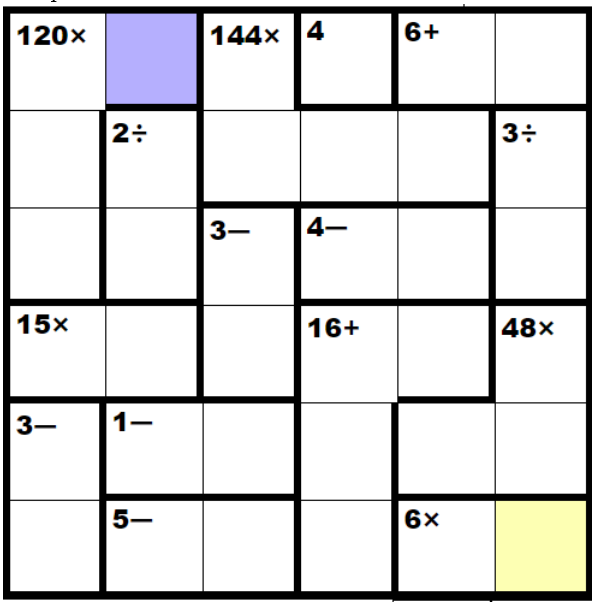
Due September 25 at the end of the lab period.

In this lab, you will write a solver for a game related to Sudoku called KenKen.

KenKen Rules

KenKen is a mathematics puzzle game similar to Sudoku, in which a $N \times N$ grid of numbers must be solved so that each square contains a number from 1 up to N. As in Sudoku, every row and column of KenKen can contain each number only once, and every number from 1 to N **must** be in each row and column. KenKen does not use the "3x3 squares" uniqueness rule of Sudoku; instead, each KenKen puzzle breaks up the squares into various irregular **"cages"**, and in each cage a single **target value** and a single **operator** is placed. When the operator is applied to the squares of the cage, the given value is obtained. The order that the operator is applied to the squares is not fixed, which is important for division and subtraction. See Wikipedia for an explanation.

Adapt the Sudoku solver from lecture to solve following KenKen puzzle:



I will give you guidelines on your solution that you must follow to complete the lab assignment. It's important to note the difficulty of debugging Prolog programs; if a mistake is made in a single clause of the program, your output will either be gibberish or the completely unhelpful "**False**". It will be important to complete this assignment in small pieces, testing each clause in an interpreter for correctness before moving on to the next task.

Representing a puzzle

Unlike Sudoku, we don't know any numbers in the initial puzzle, so we can't ask the user to provide a starting point. Instead, we'll be responsible for generating the entire solution to a puzzle that is described in terms of its cages. Your top-level `solve` functor will take two arguments: a list of cages (defined below), and a variable `G` that you will unify with the solution to the cages.

Each cage will be defined using a `cage` functor with three arguments:

1. an **atom** defining the cage's **operator**: either `add`, `mult`, `sub`, `div`, or `id` (a one-square cage without

an operator).

2. a **value** equal to the number in the upper-left corner of the cage.
3. a **list of cell coordinates** for the individual cells in the cage. A single *coordinate* will be represented as a list of two integers corresponding to a row and column position in the puzzle. For example, the list `[0, 5]` represents the upper right corner of the puzzle.

For example, the cage in the upper-left corner of the puzzle would be represented in Prolog as `cage(mult, 120, [[0, 0], [0, 1], [1, 0], [2, 0]])`. At some point you will need to define the entire puzzle in terms of its list of cages, which will be one of the two arguments you will pass to your completed `solve` functor. (The second will be a variable `S`, which `solve` will fill in with the solution.)

Solving the puzzle

Your `solve` will take a list of cages and a variable `S` and be true only if `S` is the solution to the 6x6 KenKen puzzle described by the cages. (We will assume a 6x6 puzzle.) These facts must be true about `S` (and you must enforce them in Prolog):

1. `S` must have 6 rows
2. Each row in `S` must be length 6.
3. Each row in `S` must only contain values from 1 to 6.
4. The entries in `S` must satisfy the cages of the puzzle.
5. Each row in `S` must contain all distinct values (no duplicates).
6. Each column in `S` must contain all distinct values.

Of these, only number 4 is structurally different from Sudoku, so you can code the rest by copying from the better Sudoku solver in the course Prolog repository.

Validating cages

A solution `S` is only valid if its 36 entries satisfy the puzzle's cages. To validate cages, we will write a functor that enforces a single type of cage operator. That functor is called `check_cage` and it accepts two arguments:

1. A cage, as defined above.
2. The solution `S`.

and depending on the operator of the cage, the functor selects the entries from `S` that correspond to the cage's **cell coordinates** and checks if those entries result in the cage's **value** when the cage's **operator** is applied to them. There are no "if" statements in Prolog, so we cannot write statements like "if the operator is +, then add up all the values"; instead, we write different clauses of the `check_cage` functor that match a single specific cage operator:

- `check_constraint(cage(add, Value, Cells), S)` will be true iff the sum of the entries in the `Cells` sum to `Value`.
- `check_constraint(cage(mult, Value, Cells), S)` will be true iff the sum of the entries in the `Cells` multiply to `Value`.
- `check_constraint(cage(sub, Value, Cells), S)` will be true iff the the difference of the **two entries in** `Cells` is equal to `Value`. Note that there will **always** be **only two coordinates** in `Cells`, and that the **order** of the subtraction is not specified.
- `check_constraint(cage(div, Value, Cells), S)` will be true iff the quotient of the **two entries in** `Cells` is equal to `Value`. Use `//` for division, not `/`.
- `check_constraint(cage(id, Value, Cells), S)` will be true iff the exact value of the **single entry in** `Cells` is equal to `Value`.

Coordinates vs. entries:

Note that a cage definition includes the **coordinates** of the cells that make up the cage, but don't specify the **values** at those positions (because they are unknown)... but to validate a cage using the `check_constraint` functor, we need to know the values of the puzzle at those coordinates. To translate a list of coordinates into a list of values at those coordinates, you will write a functor `cell_values(Cells, S, Values)`, which is true iff `Values` is a list of integers that corresponds to the entries of `S` at the coordinates specified in `Cells`. **For example**, if we have a 2x2 puzzle `S` whose first row contains the values `2 1`, then `cell_values([[0, 0], [0, 1]], S, Values)` would instantiate `Values` as the list `[2, 1]`.

You must write this functor. It is recursive: the cell values of an empty list of cells is empty; the cell values of a list with at least one coordinate is a list that contains `X` followed by the cell values of the tail coordinates, where `X` is the value found at the row and column specified by the head coordinate. The Prolog functor `nth0` can help here: it takes an index, a list, and a value, and is true iff the value is found in the list at the given index. Example: `nth0(1, [5, 4, 3], 4)` is true; `nth0(2, [5, 4, 3], X)` gives `X=3`; and `nth0(0, [[5, 4, 3], [2, 1, 0]], Row)` gives `Row=[5, 4, 3]`.

This functor will be necessary in all your `check_constraint` implementations. As an example usage, consider `check_constraint` for an `add` cage: to validate the constraint, we translate the `Cells` of the cage into their corresponding `Values` using `cell_values`, then see if the sum of those `Values` equals `Value`.

Sums and products:

We did a lecture example on finding the sum of a list; you will need to incorporate that functor, and then mimic it as a functor to find the product of a list. You **don't** need functors for the quotient or difference of a list of values, since division and subtraction **always** involves exactly two cells. You still need to get the values of the cells and then decide if one ordering or another of the values results in the desired difference or quotient. Recall that the `;` operator means "OR" in Prolog, as long as you remember that "OR" typically has lower priority than "AND" (the `,` operator).

HOWEVER – AND THIS IS IMPORTANT – the library we import to use `ins` and `all_distinct` does **not** use `is` for equality; it uses `#=`, as in `X #= Y + Z`. You must do the same or else Prolog will complain about variables not being sufficiently instantiated.

Checking all cages:

Once your individual `check_constraint` functors are working, you can write `check_cages(Cages, S)`, which is true iff all the cages are satisfied by the values of `S`. This is another recursive function: an empty list of cages is satisfied by any `S`; a list with at least one cage is satisfied by `S` iff the first cage is correct, and checking all the remaining cages is also correct. This is the functor you will need to call to validate #4 in the "Solving the puzzle" steps.

Recommended approach

I recommend you develop this assignment in small steps, testing each step in the Prolog interpreter before moving to the next.

1. Read the entire specifications thoroughly and make sure you understand each portion.
2. Write `sum_list` and `product_list`, which find the sum and product of a list of integers respectively. Test them in the interpreter: `sum_list([1, 3, 5], S)` should give `S=9`.
3. Write `cell_values`. This functor requires a list of coordinates and a matrix of values; invent your own small 2x2 puzzle and test this function with it.
4. Write `check_constraint(cage(id, Value, Cells), S)`. There will only be one coordinate in `Cells`; use `cell_values` to get its entry, and make sure it is equal to `Value`. Test this by using your 2x2 puzzle as `S`, and a cage that only includes one cell.
5. Write `check_constraint(cage(add, Value, Cells), S)`. Test it with your 2x2 puzzle, using cages with various amounts of cells.

6. Write `check_constraint(cage(sub, Value, Cells), S)`. You will always have 2 coordinates in a `sub` constraint.
7. Write the remaining constraint functors. Test each.
8. Write `check_cages(Cages, S)` recursively. Test it using your 2x2 puzzle and a list of 2 or more cages of different types. Test it again with a 6x6 puzzle and 2 or more cages.
9. Write `solve(Cages, S)` by following the outline in "Solving the puzzle". You will force the size of `S` to be 6x6 using the `length` functor built in to Prolog, e.g., `length(S, 6)` forces `S` to have 6 rows.

CECS 424H

You must complete this addition if you are enrolled in CECS 424H. Rewrite parts of the application so that the size of the puzzle is not assumed to be 6x6. `solve` must take an additional parameter for the size of the puzzle (the puzzle will always be a square); you must force the solution to be that size, and the numbers in the solution to only go from 1 to the size.

Deliverables

Turn in the following when the lab is due:

1. A printed copy of your code, **printed from your IDE when possible.** If you cannot print from your editor, copy your code into Notepad or another program with a fixed-width (monospace) font and print from there.
2. A copy of the query you issued to the Prolog interpreter to solve the KenKen puzzle on the first page, and the output of that query (the solution to the puzzle).