Homework 5:
Puzzling Prolog

Due: May 7 by 11:59 PM.

You can work on this assignment with one partner. You will make a single submission for the pair.

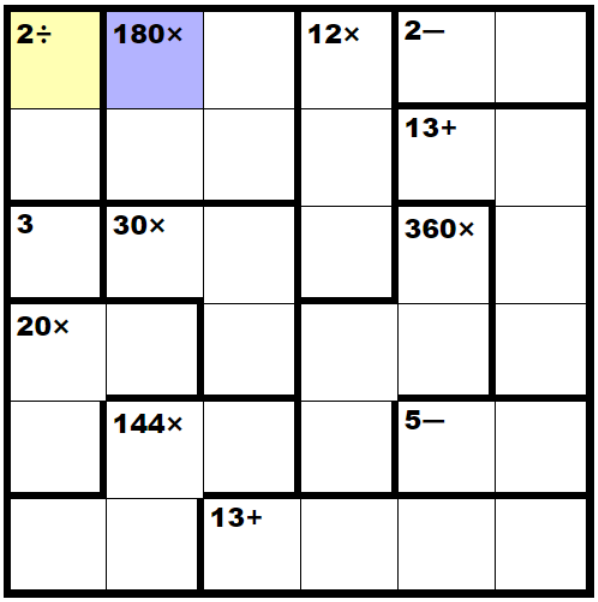
Introduction

In this assignment, 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 **target 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 variable `S` and a list of cages, and will 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. The solution `S`. (a 6x6 matrix of integer values.)
2. A `cage`, as defined above.

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(S, cage(add, Value, Cells))` will be true iff the sum of the entries in the `Cells` sum to `Value`.
- `check_constraint(S, cage(mult, Value, Cells))` will be true iff the sum of the entries in the `Cells` multiply to `Value`.
- `check_constraint(S, cage(sub, Value, Cells))` 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(S, cage(div, Value, Cells))` will be true iff the quotient of the **two entries in** `Cells` is equal to `Value`. Use `//` for division, not `/`.
- `check_constraint(S, cage(id, Value, Cells))` 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]`.

Before you can write `cell_values`, you must write a similar functor that maps a **single** coordinate pair to its value in the solution. This function, `get_cell(S, [I, J], Val)` is true iff row `I` column `J` in `S` is equal to `Val`. 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]`. (**Hint**: and what would `V` be if we next did `nth0(1, Row, V)` ? Which element in the example 2x3 matrix is that?)

With `get_cell` complete (*and tested*), you can now write `cell_values(Cells, S, Values)` as a **single statement**. How can this be? A simple observation: to construct `Values`, we must **transform** the coordinates in `Cells` into the individual values at those coordinates, producing a new list of ***transformed*** values. Do you recall a function from lab that transforms a list using a transform function?

I hope you just said `maplist`!

Sums and products:

`plus` and `times` cages can involve 2 or more cells whose values must add (or multiply) to the target. We have seen how summations and products can be written as **folds**. To implement `check_constraint(S, cage(add, Value, Cells))`, you **must** use `foldl` to check the sum of the values in the solution specified by `Cells`.

Does this sound tricky? It's not too bad.

First, you will need to define two simple functors:

1. `adds_to(X, Y, Z)`: true if Z equals X + Y; **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.
2. `mults_to(X, Y, Z)`: similarily, if Z equals X * Y.

To implement `check_constraint`, use `cell_values` to turn the `Cells` (list of coordinates) into `Values` (a list of integers from the matrix). Next, use `foldl` along with `adds_to` or `mults_to` to find the sum/product of those `Values`, which must equal the `Value` parameter to `check_constraint`.

Quotients, Differences, and IDs: Since the `sub` and `div` cages **always** involve exactly two cells, their `check_constraint` is easier. Get the `cell_values` for the `Cells`, which you **know** will be a list of two answers. You simply need to 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 precedence* (priority) than "AND" (the `,` operator).

`id` is even easier. The result of `cell_values` should be a list of only one element: `Value`.

Checking all cages:

Once your individual `check_constraint` functors are working, you can write `check_cages(S, Cages)`, which is true iff all the cages are satisfied by the values of `S`. This is another chance to apply `maplist`, this time with only two parameters: the transform function to call, and the list of cages; this form of `maplist` is true only if all values in the given list result in true/success when passed to the transform function.

What is the transform function? It should be something that will check an individual `cage` to see if its constraint is satisfied for the given solution `S`. Hmmm...

(**Hint**: remember that you can partially-apply a function to specify the first parameter, and leave the rest to be filled in later. Review the second Pokemon lab challenge.)

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(S, cage(id, Value, Cells))`. Test this by using your 2x2 puzzle as `S`, and a cage that only includes one cell.
5. Write `check_constraint(S, cage(add, Value, Cells))`. Test it with your 2x2 puzzle, using cages with various amounts of cells.
6. Write `check_constraint(S, cage(sub, Value, Cells))`. You will always have 2 coordinates in a `sub` constraint.
7. Write the remaining constraint functors. Test each.
8. Write `check_cages(S, Cages)` 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(S, Cages)` by following the outline in "Solving the puzzle".

Deliverables

Turn in the following when the lab is due:

1. A copy of your code.
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). **Make sure the entire puzzle solution is printed**; sometimes Prolog will print lists with "..." if it thinks there isn't enough room to fit the output. I want to see every single value in the lists.