Exercise 8: Solving Sudoku

This exercise wraps up our discussion of the ambiguous choice operator by getting you to build on the simple backtracking example from lecture to a full-fledged implementation of some basic AI strategies for solving **Sudoku puzzles**. *Please find the rules for Sudoku here.*

This exercise requires a good grasp of the work we've done on the choice operator to date. *We strongly recommend completing this week's lab before attempting this exercise.*

Deadline: October 12, 2019 before 10:00pm

Starter code

- `streams.rkt` (from lecture)
- `amb.rkt` (from lecture)
- `ex8.rkt`
- `p096_sudoku.txt` (sample Sudoku problems from https://projecteuler.net/problem=96)

Task 0: Brute force algorithm by combining choices

We saw in lecture how using the ambiguous operator `-<` allowed for the elegant representation of expressions comprised of multiple choices e.g. `(do/-< (list (-< 1 2 3) (-< 1 2 3) (-< 1 2 3)))`. Using choice expressions in this way produces a *Cartesian product* over the individual choices; that is, produces all possible combinations of choices. This is very useful as a tool for expressing complex combinations concisely, but actually quite poor if we want fine-grained control over how we make these choices.

Open the starter code in `ex8.rkt`, and read through the first two sections of code ("Sudoku Modeling" and "A Brute Force Algorithm"). **Do not change any code in these sections.**

Pay special attention to `brute-force-helper`, which uses recursion to make a new choice of number from 1-9 for *every* blank cell in the Sudoku board. While the code is relatively straightforward, it's not very efficient. With just 10 blank cells, this algorithm would make up to 9^10, or roughly **five billion**, different choices!

At the bottom of the file we've provided some code you can use to run the different Sudoku solving algorithms you'll develop. Try uncommenting the `(solve-brute-force easy)` call and run the file. While it does find a solution, this algorithm makes a lot of unnecessary guesses. Your task on this exercise will be to improve upon it.

(If you want to make the algorithm behave really poorly, open the Sudoku puzzle file and change one of the early 9's into a 0. Why is this so bad?)

Task 1: Narrowing choices using constraints

This brute force algorithm only checks for whether a Sudoku board is complete after all blank cells have been filled. One of the reasons the brute force algorithm is so inefficient is that it always chooses a number from 1-9 for each blank cell, and so can make early choices that are guaranteed to be incorrect, but not actually detect those choices until much later in the program execution.

One way to address this problem is to compute the choices for each cell *dynamically*, given the current state of the board. For example, in the Sudoku puzzle below, the first blank cell (to the right of the 3) can only be in the set {1, 2, 4}, since all the other numbers appear in the same row, column, or 3x3 subsquare as the cell—there is no reason at all to try any of the other numbers.

Your task here is to complete `solve-with-constraints` using this idea. You should be able to run your code on the `easy` board and find a solution by making only a single choice for each blank cell (indeed, each cell in the "easy" board has only one possible value that can be computed from looking at its row, column, and/or subsquare).

Note: Make sure that you have looked at all the helper functions that we provided. This task is not meant to be onerous, and should be very straightforward. If you decide to attempt Task 2, you should be spending much more of your time on Task 2.

Bonus Task 2: Greedily ordering choices (1% bonus)

If you try running `solve-with-constraints` on the `harder` board, you should see that it is able to find a solution, but makes quite a few incorrect choices, requiring backtracking. Your task here is to improve on this algorithm in `solve-with-ordered-constraints`, which dynamically chooses which blank cell to fill next based on the number of possible choices. In the best case, this algorithm finds that a cell only has *one* possible value (as in the `easy` puzzle); setting these cells first greatly reduces the number of remaining choices to make.

In fact, you should be able to solve all the Sudoku puzzles in the sample file using your algorithm very quickly! (Contrast this with `solve-with-constraints`...)

Recommended implementation strategy:

1. Implement `initialize-constraints`. You might want to review the higher order list functions `map` and `filter`.
2. Implement `with-ordered-constraints-helper`. This function will look similar to the other `*-helper` functions. Assume that you have a working version of `update-constraints`.
3. Implement `update-constraints`. Review the list of helper functions at the top of the file before writing any code. A new helper function might be helpful here. You'll also find the function `set-remove` helpful.
4. Implement `solve-with-ordered-constraints`. Start by writing down all the expression that you think you might need, then figure out how to put them together.