## Exercise 8: The Ambiguous Choice Operator

In the past two weeks of lecture, we've looked at an implementation of the *ambiguous choice operator* `-<`, and some applications of it. In this exercise, you'll do some more work towards applying the choice operator to interesting problems.

### Starter code

Download these files from Quercus files.

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

### Submission instructions

Submit the file `ex8.rkt` to MarkUs: https://mcsmark.utm.utoronto.ca/csc324f20/

You may not change `stream.rkt` and `amb.rkt`. You won't be able to submit those files since we'll use our own for testing purposes.

Please note that we *are* exporting (and testing) some of the helper functions in task 2, and not just the main algorithms. So please make sure to read and follow the specifications carefully for *all* functions here!

But before you submit, check that:

- Your code does not have any syntax error. If you only completed part of the exercise, replace any possibly syntactically incorrect code with the starter code.
- The `(provide ...)` code in your Racket file is left intact. Otherwise we won't be able to run our tests!

### Task 1: Making change [5 pt]

Let's look at one application of choice expressions to solve a famous problem in computer science: finding combinations of coins whose value equals a given amount.

Your first task here is to complete `make-change`, which yields combinations of 1's and 5's that sum to a given number. As you might expect, this is a choice function, meaning it returns one choice; the remaining choices are accessible by calling `next`:

```
> (define g (make-change 10))
> (next! g)
'(5 5)
> (next! g)
'(5 1 1 1 1 1)
> (next! g)
'(1 1 1 1 1 1 1 1 1 1)
> (next! g)
'DONE
```

Your next task is to generalize this to a function `make-change-gen` so that it takes two arguments: a list of coin values, and a target number.

**Note**: You might see strange behaviours when calling functions inside an `-<` expression, where the function also uses `-<` to generate choices. The workaround is to

1. First generate the choices (e.g. `-<` generates the arguments to the function that you would like to call), for example in a `let*`.
2. Call `-<` using the already generarted arguments.

The strange behaviour occurs when functions used in expressions inside `-<` uses `(fail)` to backtrack.

For example, if `f` uses `-<` and sometimes calls `(fail)`, you should *not* write:

```
(-< (f 3 5) (f 4 6))
```

Instead, choose the argument to `f` first, and then apply `f`:

```
(let* ([choice (-< '(3 5) '(4 6))])   (apply f choice))
```

**Task 2. Sudoku [5 pt]**

This task 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.*

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.

Start by reading through the functions in `ex8.rkt` under "Task 2". We provide a lot of helper functions to you, so read what they do carefully. Read through the functions `solve-brute-force` and `solve-brute-helper`. **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 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?)

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).

Once again, make sure you look at all the helper functions we provide. If you find yourself writing many functions to work with the sudoku board, you probably missed some of the functions we wrote for you. Also, we will be testing the helper functions, so make sure that their implementations follow the specifications we provide!