98-008 Homework 2: Sudoku Solver

Cooper Pierce `cppierce@andrew.cmu.edu`
Jack Duvall `jrduvall@andrew.cmu.edu`

Fall 2022

Overview

The goal of this assignment is to give you a light introduction to how closures work in Rust, by having you write a simple continuation-based Sudoku solver.

Sudoku

If you don't already know what Sudoku is, please read `https://en.wikipedia.org/wiki/Sudoku`.

Provided Datastructures

We've set you up with a couple structs and enums in `src/types.rs` that you will be using extensively; reading that file or the output of `cargo doc` is recommended. For convenience, the most important parts of this documentation are repeated here:

Number

A `Number` is a value ranging from 1 to 9. Rust doesn't yet have dependent types, so we instead use an `enum` with appropriate variants to represent each of the values. This type is `Copy`.

Two functions you may want to use on `Number`s:

- `Number::iter()`: returns an iterator over all `Number` variants, starting at `Number::One` and counting up.
- `Number::next(self) -> Option<Number>`: returns the number coming after the one passed in, if there is any

Index

A named (row, col) pair of `Number`s representing an index on a `Board`.

Like `Number`, this also has a convenient `Index::next(self) -> Option<Index>` function for getting the "next" index after any given one, going in row-major order, if there is any.

Square

An `enum` representing a single square in a `Board`. There are three variants:

- `Square::Unfilled`
- `Square::Fixed(Number)`
- `Square::Guessed(Number)`

The difference between `Fixed` and `Guessed` is that `Fixed` numbers were given with the initial puzzle and cannot change, while `Guessed` numbers are subject to change.

Board

A `struct` representing the current state of the board. Internally, it consists of a doubly-nested array of `Square`s representing all the numbers that have been placed, and a couple bitfields for efficient lookup of the sudoku constraints on those squares. The board state can only be modified through associated functions, so it will always be valid! Here are those associated functions:

- `Board::place(&mut self, Index, Number) -> bool`: attempts to place a number at a given index, returning false if it cannot. Panics if the index contains a `Square::Fixed` value.
- `Board::unplace(&mut self, Index)`: attempts to remove a placed number at a given index. Panics if the index contains a `Square::Fixed` value.
- `Board::is_filled(&self) -> bool`: returns true iff every square on the board is `Fixed` or `Guessed`.

In addition to these functions, we have a handy trait implementation that lets you do things like `board[index]` to read the `Square` at that index (but not modify it!).

What You Will Write

You will be writing code that runs the main solver recursion, building on existing datastructures given to you. This comprises editing the function `solve`, defined in `src/sudoku.rs`. This function contains comments explaining the suggested algorithm used in the refsol.

To test your code you can use the provided sudoku board files, as well as writing your own. We've provided a `main` function which wraps your code, reading from a board file passed on the command line, pretty-printing the board outputs. Like before, we've also incorporated some of the testing features in rustc/Cargo so you can automatically test your code on the provided files. Running `cargo test` will run these tests, which you can see at the bottom of `src/main.rs`.