Project 3 – Sudoku

- Issued: Tuesday, October 10
- To Be Completed By: Tuesday, October 31, 6:00 pm
- Code to load for this project:
  - A link to the system code file `sudoku_ft06.scm` is provided from the Projects link on the course web page.

Purpose

Project 2 focuses on the use of procedures, together with data structures. You will also further develop and demonstrate your ability to write clear, intelligible, well-documented procedures, as well as test cases for your procedures. In this project, we ask you to take on more of the design and implementation of sets of procedures, rather than filling in templates provided by us.

Background

Alyssa P. Hacker has become enamored with the current rage, Sudoku, so much so that she can often be seen working on Sudoku puzzles during lecture, or online at her favorite site: [http://www.websudoku.com](http://www.websudoku.com). This is a new (well, sort of) puzzle that has become very popular in recent years, and involves filling numbers into a grid of squares, subject to some interesting constraints. An example of a Sudoku grid is shown below.

![Example Sudoku Grid](https://via.placeholder.com/150)

Wikipedia characterizes Sudoku as follows:
**Sudoku,** also known as **Number Place** or **Nanpure,** is a logic-based placement puzzle. The aim of the puzzle is to enter the digits 1 through 9 in each cell of a 9×9 grid made up of 3×3 subgrids (called "regions") so that each row, column, and region contains exactly one instance of each digit. A set of clues, or "givens", constrain the puzzle such that there is only one way to correctly fill in the remainder.

Completed sudoku puzzles are a type of **Latin square**, with the additional constraint on the contents of individual regions. **Leonhard Euler** is sometimes cited as the source of the puzzle based on his work with Latin squares.

The modern puzzle **Sudoku** was invented in **Indianapolis** in 1979 by **Howard Garns**. Garns' puzzles appeared in **Dell Magazines**, which published them under the title "**Number Place**". Sudoku became popular in **Japan** in 1986, when puzzle publisher **Nikoli** discovered the game in older Dell publications. The puzzles became an international hit in 2005.

The basic idea is to try to figure out what value (from 1 to 9) to enter in each cell, so that each number appears exactly once in each row, column and region. For example, the zeroth (assume like lists we start counting from 0) and second columns each have an 8 in them, one appearing in the zeroth region, and one in the third region (note that the regions are bounded by the darker lines in the example and we are counting from the top left, going across then repeating with the second set of regions and so on). Thus we can infer that the 8 in the first column (again assuming we start counting at zero) must be in the sixth region (the one at the bottom left), otherwise either the zeroth or third region would have two 8’s in it. But one cell in that first column in the sixth region is already occupied (by a 6), and the eighth row already has an 8 in it elsewhere. So we can thus deduce that the cell at the intersection of the first column and seventh row must be an 8.

We can continue with this sort of reasoning to try to complete the puzzle. If you want, try completing this puzzle yourself, as it will give you an idea of the kinds of strategies needed to solve the puzzle (since trying all possible substitutions is horrendously expensive).

As you can see from the description from Wikipedia, Sudoku is perhaps older that Alyssa appreciates, dating back to the great Swiss mathematician Euler (though he probably didn’t treat these challenges as a puzzle). Alyssa likes solving Sudoku puzzles as an intellectual entertainment. She decides that she would like to write a program to help her crack these puzzles, though not completely solve them, that is, she would like some hints on how to proceed, and some help in filling in obvious solutions, but she recognizes that writing a program to completely solve Sudoku will be hard. (In fact, it has been shown that Sudoku is NP-complete – a technical term meaning that it is not possible to write a program that will always find a solution in a non-exponential amount of time). You are going to help Alyssa with her quest.
Infrastructure for the Sudoku system

We start by setting up a way to represent information in a Sudoku grid. We are going to do this creating a representation for a cell (or a single element in a Sudoku grid). We choose to do this by using a tagged list, where the remainder of the list will be used to represent the set of values that are still possible solutions for that cell. Here is some code to do this where each cell can take on the values 1 through 9

(define *size* 9)

(define (generate-interval a b)
  ; create list of integers between a and b
  (if (> a b)
    ()
    (cons a (generate-interval (+ a 1) b))))

(define (make-cell)
  ;; make an instance of a cell, with values from 1 to size of grid, plus label
  (cons 'cell (generate-interval 1 *size*)))

Thus, the constructor `make-cell` will create a list of values, with a tag as the first element, in this case the numbers 1 through 9 to indicate that these are all possible values to associate with this cell. To group these cells together, we start with a row of the grid. A row will be a list of cells, and the set of rows will be a list of lists of cells:

;; to start building the grid, create a row of cells of desired size

(define (make-row n)
  (if (= n 0)
    ()
    (cons (make-cell) (make-row (- n 1)))))

;; to continue building the grid, create desired number of rows of desired size
;; make-rows returns a list of rows, each of which is a list of cells

(define (make-rows n)
  (if (= n 0)
    ()
    (cons (make-row *size*) (make-rows (- n 1)))))

Thus, we can make a temporary version of a grid (meaning we are going to replace this with a more formal version shortly) by calling

(define test-rows (make-rows *size*))

You might imagine that we could just replicate this idea to create columns of the grid (vertical collections of cells, since rows represent horizontal collections of cells) and to create regions of the grid (subgrids of 3 by 3 cells). However, we want our representation for each cell to be unique (so that when we change it, or mutate its value, that change appears in the corresponding row, column and region). If we are making new cells as we create our columns, we will not have a unique representation. Another way of saying this is that our set of cells, which we create when we create
rows as lists of cells, are unique, and that rows, columns and regions are just different ways of grouping these cells together.

Hence, we need to construct some columns, where each column is a list of the actual cells from which we constructed the rows.

Here is a partial set of code to do this:

\[
\begin{align*}
&\text{(define (make-column n grid)} \\
&\quad ;; \text{given a grid represented as a list of rows, need to collect} \\
&\quad ;; \text{elements of each row into a column} \\
&\quad ;; \text{n specifies which element to collect, starting from 0} \\
&\quad \text{(map FILL-IN-HERE grid))} \\
&\text{(define (make-columns grid)} \\
&\quad \text{(define (helper n grid)} \\
&\quad \quad \text{(if (= n *size*)} \\
&\quad \quad \quad '()) \\
&\quad \quad \quad (cons (make-column n grid) (helper (+ n 1) grid)))} \\
&\quad \text{(helper 0 grid))} \\
\end{align*}
\]

**Problem 1:**

In the code you have downloaded for this project, replace FILL-IN-HERE with an appropriate expression so that calling \(\text{(make-column n grid)}\) will create a list of cells, by selecting the nth cell from each element of the grid (where a grid is a list of rows). For example:

\[
\begin{align*}
&\text{(define test-rows (make-rows *size*))} \\
&\quad ;; \text{create a set of rows, represented as a list of lists of cells} \\
&\text{(define test-column (make-column 0 test-rows))} \\
\end{align*}
\]

should provide a list of cells, that correspond to the first (or zeroth) cell of each row. You can check that your solution works correctly by mutating one of the cells of the test column and seeing what happens. For example, look at the value associated with \text{test-column} and \text{test-rows}. Now evaluate the following

\[
\begin{align*}
&\text{(set-cdr! (car test-column) '(1))} \\
\end{align*}
\]

What happens when you look at \text{test-column} and \text{test-rows}? You should see a change in the first element in each data structure (we have mutated a shared structure, which causes this to happen!).

**Problem 2:**

We need to similarly create regions. We want to represent the set of regions as a list of (nine in this case) regions, each of which is a list of cells. We choose to order the regions so that the first region in the list is the top left region, followed by the middle region of the top of the grid, followed by the right region of the top of the grid, and so on. A region we will choose to represent as a list of cells (just like
a row or a column). For a region, the list of cells should be ordered in the same way as the regions themselves, that is, the first cell in the list representing a region should be the top left cell, followed by the top middle cell, top right cell, then the leftmost cell of the middle and so on. We have provided a template for doing this:

```
(define (make-regions grid)
  (if (null? grid)
      ()
      (append (process (car grid) (cadr grid) (caddr grid))
              (make-regions (cdddr grid)))))
```

where a grid is a list of rows, each of which is in turn a list of cells.

Write the procedure `process` and any other supporting code. Once you have written your code, you can use the code provided to make a complete grid:

```
(define (set-up-game-grid)
  (let ((rows (make-rows *size*)))
    (list rows (make-columns rows) (make-regions rows))))
```

;; The following is a global variable that refers to the rows, columns and regions of a grid
;; it is not necessary but you may find it useful for debugging purposes
(define *game-grid* (set-up-game-grid))

;; here are some accessors for a game grid
(define (get-rows grid)
  (car grid))

(define (get-columns grid)
  (cadr grid))

(define (get-regions grid)
  (caddr grid))

You should be able to check your code by mutating an element of the regions representation and observing the corresponding change in the rows and columns.

## Running the Sudoku system

Now that you have completed the infrastructure to represent a Sudoku grid, we can try playing the game. We have provided a set of initial constraints (or values for specific cells) for a fairly simple Sudoku game, and some simple code to allow you to display the grid and to update values. You should look over this code to be sure you understand what it is doing, as you will be creating modified versions of the code later in the project.

Try running
If your code is correct, the system should display a grid with some values filled in, then prompt you for a command. For now, you have two choices. Entering the letter “q” will quit out of the play loop. Entering the letter “s” will allow you to specify a cell that you want to change by giving the row (which for convenience we number starting with 0 at the top of the grid) and the column (which we number starting with 0 at the left of the grid) and the new value. Try playing the game a bit to see what happens.

Problem 3:
You may notice that if you make a mistake, you are stuck. There is no way to undo a decision in the current system. Create a new version of the system that allows backtracking. This means that you should:

- Create a new version of the play interface (call it something like `play-with-backtrack`) that has an additional internal state variable, called `commands`, which is initially empty. That argument should be passed on to a new version of the driver loop.
- Create a new driver loop, which takes both a set of initial values and a set of commands. This driver loop should also recognize a new command (e.g. “b” for backtrack). When it receives this command, it should undo the last command. One easy way to do this is to create a new grid, initialize the grid, and then execute all but the last command in the list of commands, and then continue. You will want to be careful about the order in which you store commands (e.g. if you just “cons” them onto the front of a list, you will want to reverse the list before repeating the commands).

Test out your new system, showing that you can backtrack.

Problem 4:
So far we simply have an interactive way of entering values into a Sudoku grid. To really get some help for Alyssa, we need to try to reason more effectively about possible solutions.

Here is a simple starting point. Our cells in our grid start out as a list of possible values, i.e. a tagged list of the numbers from 1 to 9. But in fact, when we enter values into a cell, either as part of an initialization of the grid, or because we have entered a value in a cell as part of the interaction with the system, this has implications for other cells. In particular, if we know that a cell in some row has a unique value (i.e., we have solved the value for that cell), then none of the other cells in that row can have that value. So we should be able to update each cell in the row by removing that value from its list of possibles.

Implement this idea. For example, given a row (as a list of cells), find the cells that have only one possible value, and collect those values. (Be careful about how you do this – for example, if you map a procedure down the elements of a row, collecting the unique value in each cell if there is one, otherwise collecting an empty list, you will end up with a list structure that needs to be further processed to create a list of unique values.)
Once you have a set of unique values for a row, process all the other cells in the row, remove these unique values from their set of values. Be sure that you mutate the tagged list that represents a cell, rather than just creating a new list!

Do the same thing for columns, and for regions.

Test your code.

Finally, modify your driver loop (or create a new version) so that when you enter a new value into a cell, you also update the rows, columns and regions. You might find it interesting to display the grid after you enter your value, then do the update, and display the grid again. Notice what happens. As you add values to the grid, in some cases, this will constrain the values in other cells to a unique value, which then reduces the puzzle.

**Problem 5:**
The code you have written will help Alyssa with solving the puzzle, since it will fill in any cells where the answer is uniquely constrained. But for a sparsely filled grid, this may not work very often. So we would like to help Alyssa further by providing some hints. Here is the idea behind one strategy for filling in cells:

- Consider a set of columns all of which are components of the same set of regions, e.g., columns 0, 1 and 2 are all components of region 1, 4 and 7; columns 3, 4 and 5 (regions 2, 5, and 8); and columns 6, 7 and 8 (regions 3, 6, and 9).
- Suppose we collect the set of unique answers contained in each column in one of these groups, one set for each column – for example, we find the set of known values in column 0, in column 1 and in column 2.
- If there is a value that appears in two of these columns but not the third, then that value is a good candidate for further examination. (This is exactly what happened in the example of page 1, where columns 0 and 2 had an 8 in them.) In particular, that value will be constrained to a specific column (the one in which it is not already a unique answer) and to a particular region (the one that does not contain that value as a unique entry). Thus, this value is now restricted to one of three possible slots, defined by the intersection of the column and the region.

Implement a new driver loop and user interface, which accepts a new command, “h” for help. If given this command, the system should analyze each set of three columns as described above, and print out information identifying possible combinations of a column, region and value that are worth considering. It should then continue the driver loop so that you can use this information to select a possible cell whose value you are going to set.

Demonstrate your code on some test cases.
Problem 6:
The solution in Problem 5 simply printed out information on possible areas on which to focus, including a set of three cells and a possible value. But we can do better. Add to your system from Problem 5, by implementing code that takes a column, a set of three rows, and a value, and checks to see if that value can only fit in one of the three cells defined by the intersection of the rows and column. A cell is a possibility only if it is not already restricted to a unique value, and if the value does not already appear as a unique value elsewhere in the row. Augment your “help” method by printing out possible cells and values using this idea.

Problem 7:
Develop some other strategy for providing help to Alyssa, and implement it as part of a driver loop. Describe your idea as well as providing the code used to implement it.

Project Submission

For each problem above, include your code (with identification of the problem number being solved), as well as comments and explanations of your code, and demonstrate your code’s functionality against a set of test cases. Once you have completed this project, your file should be submitted electronically on the 6.001 on-line tutor, using the Submit Project Files button. Please be sure that you save your definitions using File/Save Other/Save Definitions as Text when you are ready to save a version to submit. Just submitting the .scm file leads to unreadable files for the TA’s.

Remember that this is Project 3; when you are have completed all the work and saved it in a file, upload that file and submit it for Project 3.