MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001 – Structure and Interpretation of Computer Programs
Spring Semester, 2007

Project 3 – Sudoku

- Issued: Monday, March 19
- To Be Completed By: Friday, April 6, 6:00 pm
- Code to load for this project:
    - A link to the system code file `sudoku.scm` is provided from the Projects link on the course web page.

Purpose

Project 3 focuses on the topics of procedures, data structures, and mutation. You will also further develop and demonstrate your ability to write clear, intelligible, well-documented procedures, as well as test cases for your procedures. One important difference in this project is that we ask you to take on more of the design and implementation of sets of procedures, rather than filling in templates provided by us. This is an important skill to develop, as in the future you will often be required to create software systems based just on general design characterizations.

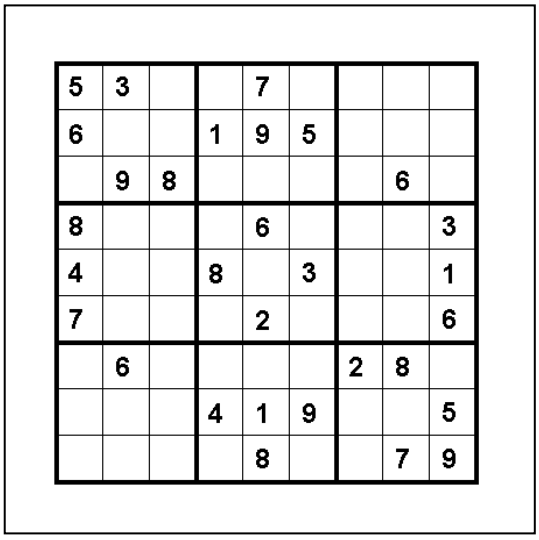
Although we said earlier in the course that all/most procedures return values, in this project we will be focusing less on return values and more on "side effects." This will become apparent when you get to the problems involving the driver loop code (discussed below), which looks very different from the other, mostly functional code we've dealt with before.

As you write code for this project, it is important to write clean code (easy to read, easy to follow). As well, your code should be as simple as possible – or at least not needlessly complex – and you should re-use existing procedures whenever possible or whenever it makes sense to do so. You may also find it good practice to use defensive programming – specifically to check for types of arguments in your code as a way of helping you debug your code, and as a way of ensuring that the correct type of data is being passed to procedures.

Be sure to read the entire project description below before you start working. You need to get a sense of what is being asked before your start planning your approach. Be sure to include your test cases, and appropriate documentation for each problem.

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 (which doesn't thrill the lecturers!), or online at her favorite site: 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.

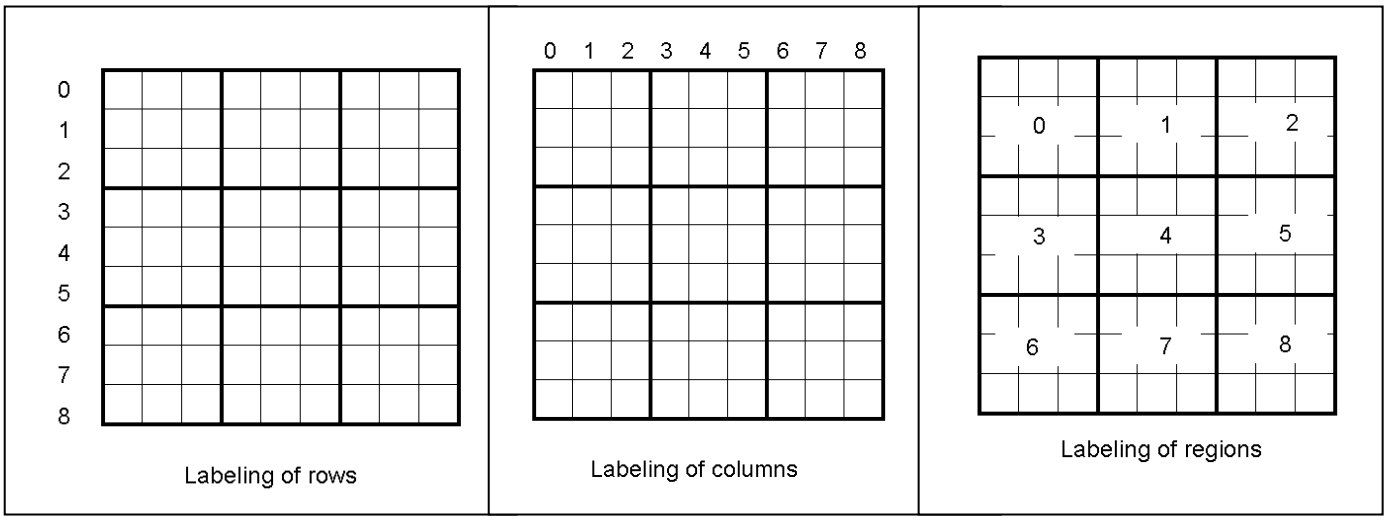


Wikipedia characterizes Sudoku as follows:

> ***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. Before we show an example, let's label the rows, columns and regions (the 3 by 3 collections bounded by the darker lines in the example). We will label the rows starting at the top, with the label 0, as shown in the figure below; we will label the columns starting at the left, with the label 0, as shown in the figure below; and we will label the regions, by moving from left to right, and top to bottom, starting with the label 0, as shown in the

figure below (within each region we will use the same ordering to refer to individual cells within a region):



If we go back to our initial grid, we can see that the zeroth (assuming 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. Thus we can infer that the 8 in the first column 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, as shown below.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
	8		4	1	9			5
				8			7	9

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 that roughly means that nobody knows how to write a program that will always find a solution in a less-than-exponential amount of time, as a function of the grid size, and many computer scientists believe that it's technically impossible. If you are interested, there is a much more formal definition of NP-complete that you will see in 6.045 and similar courses). 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 by 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 tag identifies the data structure as a cell, and the remainder of the list will be used to represent the set of values that are possibilities for the solution for that cell. The idea is that we will start with a set of possibilities for the correct value in a cell, and as we gain more information, we will reduce that set of possibilities for cells, until there is only one value left (which we refer to as **unique** and which represents the proposed value for a cell as part of the solution of the entire grid. We also need to decide on the size of the grid. This may sound odd, since in the examples above (and any Sudoku puzzle you may have solved) the size is 9, but in fact the concept holds for any perfect square, so we could have grids of size 4, 9, 16, and so on. You will probably find that it is easier to initially debug your code using the smaller grid of size 4, before moving on to the standard grid of size 9. Hence we provide a global variable:

```
; Size of the grid.  Must be a perfect square (4, 9, 16, etc.)
; Initialized to 4 to make debugging simple.
(define *size* 4)
```

A cell has a tag, as well as a set of possible values (notice the new data type `Cell` and its definition). A cell also has an accessor for retrieving the set of possible values and a mutator for changing the set of possible values.

```
;;; Cells

; Cell = Pair<tag-symbol, List<int>>

(define (make-cell)
  ; makes an instance of a cell, with values from 1 to *size*, with tag
  ; type: -> Cell
  (cons 'cell (generate-interval 1 *size*)))

; accessors
```

```
; get possible values for the cell
(define (cell-values cell) (cdr cell))


; mutators

; set possible values for the cell
(define (set-cell-values! cell values)
    (set-cdr! cell values))


(define (generate-interval a b)
  ; create list of integers between a and b
  ; type: int,int -> list<int>
  (if (> a b)
      '()
      (cons a (generate-interval (+ a 1) b))))
```

Thus, the constructor `make-cell` will create a list of values, with a tag as the first element, and the numbers 1 through *size* to indicate that these are all possible values to associate with this cell. Note that `generate-interval` is one of several useful list procedures that we provide in the code.

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

;;;;;;;;;;;;;;;;;;
;;; Rows

; Row = List<Cell>

(define (make-row n)
  ; makes a row of n cells
  ; type: int -> Row
  (if (= n 0)
      '()
      (cons (make-cell) (make-row (- n 1)))))

;; to continue building the grid, create desired number of rows of desired size

(define (make-rows n)
  ; make a list of n rows
  ; type: int -> List<Row>
  (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, which are 3x3 cells for the usual Sudoku grid, or 2x2 cells if we're using the size-4 grid). However, we want our representation for each cell to be unique (so that when we change it, or mutate its value to keep track of possible values for that cell, 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, since the version of the cell in a row will be a different data object from the version of the cell in a column. In other words, our set of cells, which we create when we create rows as lists of cells, is unique, and 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 – you will be filling in some of this as part of the project:

```
;;;;;;;;;;;;;;;;;;
;;; Columns

; Column = List<Cell>

;; given a list of rows, need to collect
;; elements of each row into a column

;; YOU WILL NEED TO FILL IN make-column (Problem 1)

(define (make-column n rows)
  ; collect the nth element of each row into a column
  ; n specifies which element to collect, starting from 0
  ; type: int,List<Row> -> Column
  (map FILL-IN-HERE rows))

(define (make-columns rows)
  ; given a list of rows, make a list of columns from it
  ; each column should share cells with each row
  ; type: List<Row> -> List<Column>
  (define (helper n rows)
    (if (= n *size*)
        '()
        (cons (make-column n rows) (helper (+ n 1) rows))))
  (helper 0 rows))
```

And we need to construct regions, where each region is a list of actual cells from which we constructed the rows.

```
;;;;;;;;;;;;;;;;;;
;;; Regions

; Region = List<Cell>

(define (make-regions rows)
```

```
  ; given a list of rows, make a list of regions from it.
  ; type: List<Row> -> List<Region>
  (if (null? rows)
      '()
      (append (make-some-regions (first-n rows (region-size)))
              (make-regions (but-first-n rows (region-size))))))

;; YOU WILL NEED TO FILL IN make-some-regions for Problem 2

(define (make-some-regions some-rows)
  ; takes a few rows of the grid (a number of rows equal to (region-size))
  ; and returns their regions.
  ; type: List<Row> -> List<Region>

  FILL-IN-HERE
)

; Size of a region.  For example, a 9x9 Sudoku grid has 3x3 regions.
; We make this a procedure rather than a variable so that the value of region-size
; always reflects the current value of *size*.
(define (region-size) (sqrt *size*))
```

Note that the size of each region depends on the size of the grid – the usual 9x9 Sudoku grids have 3x3 regions, while 4x4 grids have 2x2 regions, and so forth. So we have a procedure `region-size` that takes no arguments and returns the size of a region for the current value of `*size*`.

Once we have completed this code (which you will do below), we can create a grid:

```
;;;;;;;;;;;;;;;;;;
;;; Grid

; Grid = List: tag, List<Row>, List<Column>, List<Region>

(define (make-grid)
  (let ((rows (make-rows *size*)))
    (list 'grid rows (make-columns rows) (make-regions rows))))

;; here are some accessors for a game grid

(define (get-rows grid)
  (second grid))

(define (get-columns grid)
  (third grid))

(define (get-regions grid)
  (fourth grid))

(define (get-cell grid r c)
  ; gets the cell at row r, column c in the grid.
  ; (counting from 0).
  ; type: Grid,int,int -> Cell
  (let* ((rows (get-rows grid))
         (row (list-ref rows r))
         (cell (list-ref row c)))
```

```
    cell))

;; mutator for game grid
(define (set-value! grid r c val)
  (set-cell-values! (get-cell grid r c) (list val)))
```

Your turn:

To get started, create a file in which you will store your solutions. Place the expression

```
(load "sudoku.scm")
```

at the top of your file (assuming you have downloaded the code for the project and saved it in your directory). This will load the code we have provided for you into your Scheme environment.

Problem 1:

We need to be able to complete the construction of a grid. First, copy the definition for `make-column` into your file. Replace `FILL-IN-HERE` with an appropriate expression so that calling `(make-column n rows)` will create a list of cells, by selecting the nth cell from each element of the grid (where `rows` is a list of rows). For example:

```
(define test-rows (make-rows *size*))
;; create a set of rows, represented as a list of lists of cells

(define test-column (make-column 0 test-rows))
```

should provide a list of cells, that correspond to 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 `test-column` and with `test-rows.` Now evaluate the following

```
(set-cdr! (car test-column) '(1))
```

What happens when you look at `test-column` and `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!). Note that in general we are going to use data abstraction selectors and mutators to access and change values – here we directly manipulate the list structure just so you can see what is happening.

Problem 2:

We need to similarly create regions. We want to represent the set of regions as a list of regions (nine of them for the usual 9x9 grid, but only four for the 4x4 grid). Each region is in turn a list of cells. We choose to order the regions so that (in the 9x9 grid) 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, described above. Copy the template for `make-some-regions` into your solution file, and complete the definition.

Once you have written your code, you can use `make-grid` to make a complete grid. Also note that `set-value!` should now be used to change the value of a cell. You should be able to check your code by mutating an element of the regions representation using this procedure, and observing the corresponding change in the rows and columns. To help you with this, we have provided a special mutator:

```
(define (set-value-row! rows r c val)
  (let ((row (list-ref rows r)))
    (let ((cell (list-ref row c)))
      (set-cell-values! cell (list val)))))
```

In order for the rest of the project to succeed, you need to be sure that your code for constructing a grid creates a representation with shared cells. To test this, do the following:

```
(define test-rows (make-rows *size*))
(define test-columns (make-columns test-rows))
(define test-regions (make-regions test-rows))
```

This creates a new set of cells, which should be shared by the column and region representations. To test this, look at the value associated with each of these variables. Then, use `set-value-row!` to change the value in a cell, e.g. `(set-value-row! test-rows 1 2 3)` and again look at the values associated with each of these variables. You should see a change in the value of a cell, and you should be able to confirm that it is the correct cell (remember that the ordering of elements is different for rows, columns and regions). Run some additional tests to check that your representation is correctly sharing cells.

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

```
(play-simple *initial-4x4-values*)
```

If your code is correct, the system should display a 4x4 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 (remember we number starting with 0 at the top of the grid) and the column (remember 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. Note that this is using a 4 by 4 grid. When you are comfortable that your code works correctly, try changing the size, and looking at a bigger Sudoku grid:

```
(set! *size* 9)

(play-simple *initial-9x9-values*)
```

Adding to the system:

The rest of this project is going to ask you to create some code to improve the system for solving Sudoku puzzles. Each problem involves designing and implementing of a set of procedures, so you should plan your approach before starting to code. Also, most of the changes will involve a new play interface `play-with-backtrack` and a new associated driver loop. Rather than creating a new play interface for each problem, we suggest that you simply use one procedure, but clearly label with comments the changes that you make in response to each problem. You should also clearly annotate with comments the test cases you use for each problem, as well as the code you wrote for each problem.

Also note that the driver loop uses commands such as `printf,` which is a Scheme procedure for printing information on the screen, in a particular format. You can read about this procedure in the DrScheme documentation, but here are some simple facts about `printf.` This procedure takes a string as its first argument, followed by an arbitrary number of additional arguments. If applied to a standard string, it simply prints that string on the display. If the keyword `~n` is used, then a new line is printed out, for example

```
(printf "~nThis is a string~n")
```

will insert a blank line, then print the string "This is a string" and then insert another blank line.

If you want to print the value of an expression, an easy way to do this is to include the keyword `~a` in the string, and then include as a separate argument the expression whose value you want to print. For example,

```
(define *size* 4)

(printf "~nThe size of the grid is ~a."  *size*")
```

would result in the following being printed on the monitor:

```
The size of the grid is 4.
```

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 want 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 *n* (where *n* is the size of the grid). 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 determined the specific 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, incorporate this new code into your driver loop, 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:

You may notice that your code from the previous problem may leave the grid in a state where there are some obvious holes unfilled. Depending on the order in which you fill in cells, for example, the scheme for updating unique values may leave a single cell in a region unfilled, a situation that clearly has only a unique answer left. This is because our updating scheme depends on the order in which we update the row, column and region information. Modify (or create a new version) of your code from Problem 4 so that the procedure that removes unique values from the set of possible values in cells in a row, column or region keeps track of how many changes are made. Your new code should keep iterating through the process of updating row, column and region information until no further changes are noted.

Test your new code.

Problem 6:

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 (i.e. suggestions of places on which to focus, as opposed to explicitly solving for a unique cell value). Here is the idea behind one strategy for filling in cells:

- Consider a group 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 0, 3 and 6; columns 3, 4 and 5 (regions 1, 4, and 7); and columns 6, 7 and 8 (regions 2, 5, and 8).
- 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, than 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.

Modify your driver loop and user interface to accept a new command, “h” for help. If given this command, the system should analyze each set of three columns as described above (columns 0, 1 and 2; then 3, 4 and 5; then 6, 7 and 8), and print out information identifying possible combinations of a column and value that are worth considering further. This means it should print out combinations of a column number and a value that have the property that the value appears in the other two columns of the group as a unique value, but not in the current column. Don’t worry about the fact that you can

actually constrain the information to a specific region, even this information is valuable; just list the column and value. The driver loop should then continue so that you can use this information to select a possible cell whose value you are going to set.

Note that this strategy is designed for 9x9 grids, which are the only grids that Alyssa likes to play. (She finds 4x4 too easy, and 16x16 too hard.) So you can implement the help command only for 9x9 grids. You don't need to generalize this strategy to 4x4 grids or other grid sizes.

Demonstrate your code on some test cases. We would suggest that you fill in a bit of the grid before testing the "help" method, since otherwise it may generate a huge number of possibilities.

Problem 7:

The solution in Problem 6 simply printed out information on possible areas on which to focus, listing columns and values that might be easy to reason about. But in fact we can do better. Add to your solution for Problem 6, to consider each possible cell in a column, checking to see if the value suggested is in fact one of the possible values at that cell. If it is, keep that information about the row, column and value that is a possible solution, if not, discard that possibility. In this way you can collect a set of (row, column, value) entries that you can then print out as advice to Alyssa. Augment your "help" method by printing out possible cells and values using this idea.

Again, this only needs to work on 9x9 grids.

Problem 8:

Develop some other strategy for providing help to Alyssa on 9x9 grids, 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.**

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.