Project 2: Sudoku

Due: Before lab starts on November 13th

1 Introduction to Sudoku

Sudoku is a logic puzzle set on a nine by nine grid. The goal is to fill in the blank spaces in the puzzle with the digits 1 thru 9, according to these four rules:

- each row contains each digit once;
- each column contains each digit once;
- each three by three sub-grid (with dark lines) contains each digit once;
- each cell contains only a single number.

To solve the puzzles, you must deduce what numbers go where, based on the positions of numbers already in the puzzle. As an example, here is a puzzle:

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

Consider the left-most column. Since it has the numbers 4, 5, 6, 7, and 8, we know that the blank spaces in that column must be filled with 1, 2, 3, and 9. Now, look at each row that intersects those blank spaces. The first one has a 9 already, as do the last two. So, the only possible place to put a 9 is in the starred cell:

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

In that same puzzle, look at this starred cell:

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

We know from the right-most column that that cell cannot have a 1, 3, 5, 6, or 9. But, since the bottom-right 3x3 grid cannot duplicate any numbers either, we know that that cell cannot be 2, 7, or 8. So, that cell must be a 4.

2 Advice & Logistics

Teachpack. The Sudoku teachpack is here: `http://www.cs.uchicago.edu/~robby/courses/15100-2007-fall/sudoku-tp.ss`. The provides all of the exports of the `image.ss`, in addition to the operations described here. Do not use any other teachpacks.

Solution submission. Email solutions (the "Intermediate Student with Lambda" code, not .pdfs or word files, etc.) to both `robby@cs.uchicago.edu` and `varsha@cs.uchicago.edu`.

3 Mechanically solving Sudoku puzzles

This section gives an overview of how your program will proceed, and the rest of this document breaks down the job down into a number of programming tasks. Read this section carefully *before* beginning to program.

Overall, you will solve Sudoku puzzles by translating them to logical formulas and then simplifying the formulas. First, you convert the rules above into a list of formulas that must all be true and to write a program that simplifies the formulas until there are no more simplifications possible.

We begin with 9^3 (729) claims. Each claim consists of three numbers: a column-number (between 1 and 9), a row-number (between 1 and 9), and a digit (between 1 and 9).

```
;; an claim is:
;; (make-claim number[1-9] number[1-9] number[1-9])
(define-struct claim (i j n))
```

Each of the claim's truth values indicates if a particular cell has a particular value. For example, if `(make-claim 2 3 4)` holds then the cell at (2,3) has a 4. If the claim `(make-claim 2 3 5)` does not hold, then the cell at (2,3) does not have a 5. The coordinates use the standard computer-science ordering: the upper-left is (0,0), increasing the first coordinate moves to the right and increasing the second coordinate moves down.

Each of the rules of Sudoku corresponds to some axioms about all Sudoku puzzles, and we can express each axiom as a list of claims where exactly one of the claims in the list must true, and the rest must be false. (Of course we do not know ahead of time which is the true claim and which are the false ones and which is which varies for each puzzle.)

```
;; an axiom is:
;; (listof claim)
```

As an example, the first rule tells us that there can only be a single number 1 in the first row. Written as an axiom, we have this:

```
(list (make-claim 1 1 1) (make-claim 2 1 1) (make-claim 3 1 1)
      (make-claim 4 1 1) (make-claim 5 1 1) (make-claim 6 1 1)
      (make-claim 7 1 1) (make-claim 8 1 1) (make-claim 9 1 1))
```

Similarly, the second rule tells us that there can only be a single number 1 in the first column, which is represented by this axiom:

```
(list (make-claim 1 1 1) (make-claim 1 2 1) (make-claim 1 3 1)
      (make-claim 1 4 1) (make-claim 1 5 1) (make-claim 1 6 1)
      (make-claim 1 7 1) (make-claim 1 8 1) (make-claim 1 9 1))
```

The third rule tells us that there can only be a single number 1 in the top-left 3x3 grid, which corresponds to this axiom:

```
(list (make-claim 1 1 1) (make-claim 1 2 1) (make-claim 1 3 1)
      (make-claim 2 1 1) (make-claim 2 2 1) (make-claim 2 3 1)
      (make-claim 3 1 1) (make-claim 3 2 1) (make-claim 3 3 1))
```

Finally, this axiom tells us that there can be only one number in the top-left cell:

```
(list (make-claim 1 1 1) (make-claim 1 1 2) (make-claim 1 1 3)
      (make-claim 1 1 4) (make-claim 1 1 5) (make-claim 1 1 6)
      (make-claim 1 1 7) (make-claim 1 1 8) (make-claim 1 1 9))
```

Of course, there are many more axioms, but these give you the flavor of them.

There are two basic rules for learning the truth value of a claim, based on the axioms.

1. If one of the claims in an axiom is true, then all of the rest must be false.
2. If all of the claims in an axiom are false except for one, then that one must be true.

So, to solve a Sudoku puzzle, we can translate the initial, given numbers from the puzzle into some claims that we know are true. Then, we can use the two basic rules to find out the truth values for all of the rest of the claims. The rest of the document gives some guidance on how to do that and how to build a stepper that allows you to watch your solver chug away.

A Representing (partial) knowledge

The first programming task is to design a data definition to represent the truth values of the 729 claims. Values of this data definition should contain the truth values (true, false, or unknown) for each of the 729 claims.

Since there are 729 of them, you might naturally imagine putting the claims into a list. But, this has a tremendous performance penalty, since looking at the value of each element of the list once takes roughly $\frac{729^2}{2}$ (265720) steps. If, instead, we used a struct that had 729 fields, looking at each of these fields once would only take roughly 729 steps, a significant improvement. Of course, a structure with 729 fields is far too complicated to be useful.

Instead, we use can use a `vector`. A `vector` is similar to a struct in that it has fields and the access to each field only requires a single step. Unlike structs, however, we access the fields of a vector by numeric indices (starting at 0), rather than by field names.

To get started, design a pair of functions that convert claims to numbers between 0 and 728, and numbers between 0 and 728 to claims:

```
;; claim->num : claim → number
;; to convert a claim to a number between 0 and 728.
(define (claim->num claim) ...)

;; num->claim : number → claim
;; to convert a number between 0 and 728 to a claim
(define (num->claim n) ...)
```

To write these functions, it helps to think of the claim as a three-digit number, written in base nine. Here is a data definition for arbitrary numbers in base nine:

```
;; a base-nine-nat is either:
;; - a digit (i.e., an integer between 0 and 8, inclusive), or
;; - (+ (* base-nine-nat 9) digit)
```

where the selectors and digit predicate are:

```
;; digit? : base-nine-nat -> boolean
(define (digit? n) (<= n 8))

;; all-but-last-digit : base-nine-nat -> base-nine-nat
(define (all-but-last-digit b9n) (quotient b9n 9))

;; last-digit : base-nine-nat -> digit
(define (last-digit b9n) (modulo b9n 9))
```

```
;; build-vector : nat (nat → X) → (vector X)
;; creates a vector of size n filling all of the
;; positions in the vector with the results of f
(define (build-vector n f) ...)

;; vector-ref : (vectorof X) nat → X
;; extracts the element at position n from the vector v
(define (vector-ref v n)

;; vector-length : (vectorof X) → nat
;; computes the number of elements in v
(define (vector-length v) ...)

;; vector->list : (vectorof X) → (listof X)
;; builds a list containing the elements of v
(define (vector->list v) ...)
```

Figure 1: Functions on vectors from the `sudoku-tp` teachpack

Now that we can convert a claim into a number between zero and 728, we can represent our current knowledge of the set of true, false, and unknown claims as a vector of 729 elements. The vector that has one element for each claim, and the element at a specific position is either a boolean indicating the claim's value is known to be the boolean, or `'unknown` to indicate it is not known:

```
;; a claim-table is:
;; (vectorof status)

;; a status is either:
;; - true
;; - false
;; - 'unknown
```

Using the functions in figure 1, define a library of helper functions on claims:

```
;; claim-true? : claim-table claim → boolean
;; indicates if the claim is true
(define (claim-true? table claim) ...)

;; claim-false? : claim-table claim → boolean
;; indicates if the claim is false
(define (claim-false? table claim) ...)

;; claim-unknown? : claim-table claim → boolean
;; indicates if the claim's truth value is unknown
(define (claim-unknown? table claim) ...)

;; mostly-unknown-table : (listof claim) → claim-table
;; constructs a claim-table where all claim are 'unknown,
;; except those in the argument claims, which are true.
(define (mostly-unknown-table claims) ...)
```

In addition to those, we need a way to build up tables that have more information about the truth or falsehood of particular claims. An `assertion` tells us the truth or falsehood of a particular claim:

```
;; an assertion is
;; (make-assertion claim boolean)
(define-struct assertion (claim true?))
```

and the `apply-assertion` function records the assertion in claim table:

```
;; apply-assertion : assertion claim-table → claim-table
;; builds a new table with the assertion a recorded in the table
(define (apply-assertion assertion table) ...)

;; a few examples as tests to get started
(claim-true?
  (mostly-unknown-table empty)
  (make-claim 1 1 1))
false

(claim-true?
  (apply-assertion (make-assertion (make-claim 1 1 1) true)
                   (mostly-unknown-table empty))
  (make-claim 1 1 1))
true
```

B Sudoku axioms and solving puzzles

The next step in solving a Sudoku puzzle is building all of the axioms like the ones in section 3.

```
;; axioms : (listof axiom)
(define axioms  ...)
```

Hint: *There should be 324 axioms in your list. You will need multiple helper functions and recursion on natural numbers to build that list.*

Once you have the list, design these functions:

```
;; find-claim : claim-table (listof (listof claim)) → assertion or false
;; find a variable that is definitely either true or false, by
;; attempting to apply the two rules of inference to each axiom
;; in axioms (the rules of inference are at the end of section 3)
;; If no more rules apply, returns false.
(define (find-claim table axioms) ...)

;; next : claim-table → claim-table or false
;; uses find-claim and apply-assertion to compute an claim-table
;; with more information than the input table, if possible
(define (next table) ...)

;; solve : claim-table → claim-table
;; solves a Sudoku puzzle by repeatedly calling next
;; until no more steps are possible.
(define (solve table) ...)
```

Hint: *Do not use the* `solve` *function until you are sure the rest of the functions are working well. It may take several minutes to complete.*

C Single-stepping Sudoku

The `solve` function is kind of boring, because it just waits silently for so long before producing an answer. We can, however, design a new solving function that shows how our solving algorithm is proceeding.

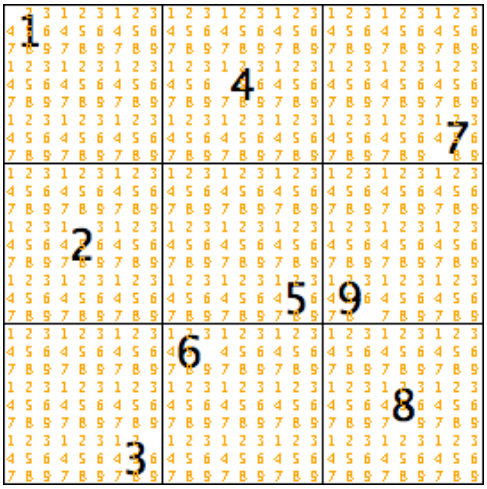
So, the goal of this portion of the exercise is to design a function that draws an `claim-table`, showing the state of the variables:

```
;; claims->image : claim-table → image
(define (claims->image claim-table) ...)
```

but this function needs a large number of helper functions.

The overall strategy for designing this function is to find all of the claims that are true and draw big, black numbers in their corresponding places on a Sudoku puzzle. Then, find all of the variables that are maybes and draw smaller, orange numbers that show where they are. Here is an example:

```
(claims->image
 (mostly-unknown-table
  (list (make-claim 1 1 1)
        (make-claim 2 5 2)
        (make-claim 3 9 3)
        (make-claim 5 2 4)
        (make-claim 6 6 5)
        (make-claim 4 7 6)
        (make-claim 9 3 7)
        (make-claim 8 8 8)
        (make-claim 7 6 9))
```



So, we can split this up into a few jobs. First, design a single image that forms the backdrop for the Sudoku board. Its size should be big enough to hold all of the big black letters or all of the small orange letters, whichever takes up more space.

Second, design a function that takes a true claim and overlays the corresponding number into the proper place in a board.

```
;; overlay-true : image claim -> image
;; to draw a big black digit in the right place on the board
(define (overlay-true board claim) ...)
```

Third, design a function that takes an unknown claim and overlays the corresponding number into the proper place.

```
;; overlay-unknown : image claim -> image
;; to draw a small orange digit in the right place on the board
(define (overlay-unknown board claim) ...)
```

Fourth, build functions that, using `vector->list`, extract the true and unknown axioms from an `at-table`. Finally combine these functions to complete `claims->image`. (Of course, the functions above require helper functions.)

Once you have defined a function that renders images, define a new version of `solve` called `solve/images` that behaves just like `solve`, except that around the recursive call in `solve/images` put `with-image` from the teachpack. This new construct `with-image` accepts two arguments. The first should be an image and the second can be anything. When the little man encounters `with-image`, he shows the image in a window and then continues with the second argument. So, if your `solve` function has a recursive call that looks like this: `(solve new-table)`, the `solve/images` function would have a recursive call with `with-image` and otherwise be the same:

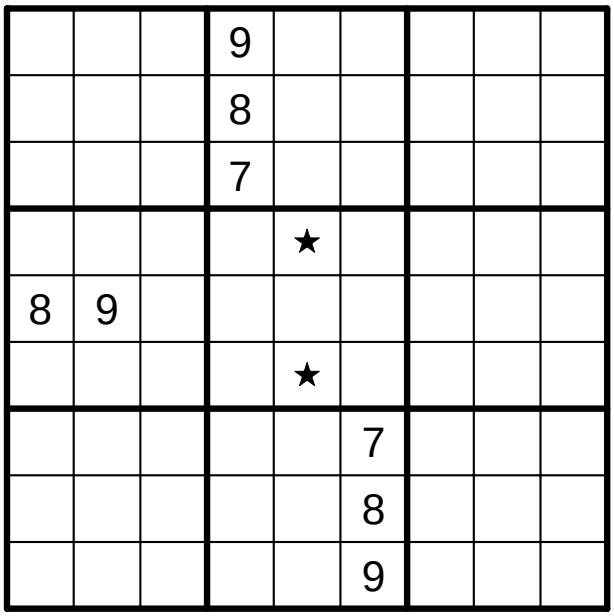
```
(define (solve claim-table)
  ... (solve new-table) ...)

(define (solve/images claim-table)
  ... (with-image (claims->image new-table)
                  (solve/images new-table)) ...)
```

D Food for thought & Performance

This section is just a bonus for those who are interested in Sudoku. If you finished all of the earlier work, you are done with the project.

Not all puzzles can be solved using the method you've implemented. As an example, consider this puzzle:



If you were to run it in your program, you would get the same puzzle back, with no new big black numbers. But, we actually know one more number.

From the 7s, 8s, and 9s in the top and in the bottom of the puzzle, we know that the center sub-grid's 7, 8, and 9 must be in the center column. We also know that the 8 and the 9 cannot be in the center row,

and thus the 8 and 9 in the center sub-grid must be in the starred cells. (We don't know which one goes in which cell, however.) This means the only possible place for the 7 in the center sub-grid is the center of the puzzle.

Performance. Do not read the rest of this section until your solver is working. You may notice that your solver is not particularly fast. Partly, this is due to all of the extra checking that the teaching languages do. Once you are sure your program is working well *and not before*, you can change the language level to make it run faster. Beware, however, that this will make your program much harder to debug; it is so bad, in fact, that some programs that should signal errors will now just give bad output. You have been warned. To speed it up, follow these steps:

1. Select the Choose Language... menu item in the Language menu.
2. Choose the Pretty Big language level, under the PLT turndown triangle.
3. Before closing the language dialog, click the Show Details button, which will make some more options visible.
4. Click No debugging or profiling.
5. Remove the teachpack.
6. Add a new first line to your program: `(require "sudoku-tp.ss")` after making sure your solution file is in the same folder as the Sudoku teachpack.

Now your solver should run much more quickly.