Thinking about Sudoku and Scheme: First Steps Towards a Solver

Paul R. Potts

January-February 2006 and March 2025

In early 2006, I wrote a series of blog posts describing my attempts to continue learning Scheme programming by implementing some code to solve Sudoku puzzles. Sudoku was all the rage at the time so it seemed like a natural thing to do. Working "in public" like this is both rewarding and a bit risky: rewarding, because if you come up with some good code, people may find it and make use of it, or jump in to the conversation. But if you don't, your failure is left hanging out there in public for all to see.

I wouldn't call this exploration a total failure, but it petered out in an unsatisfying way. The original blog posts got no comments. I did not really get a finished solver before I had to move on to other paid work — the code I completed could only take some simple first steps towards solving a puzzle. I don't feel that I found the right abstractions that would allow me to best express additional puzzle-solving strategies. But I did better learn Scheme.

I also got a lesson in how plain R5RS Scheme, a stripped-down dialect of Lisp with minimalist data structures and standard libraries, may be great for implementing other Lisp-like languages, but was probably not the best language to use for this purpose. In fact one can see the development of Haskell as a reaction, by the functional programming community, to the limited support for higher-order functional programming concepts in Lisp and Scheme. After learning some Haskell, R5RS Scheme, without many "batteries" included, feels limited in expressiveness and tedious to debug. So, I have not felt motivated to chew on this Scheme implementation further. It's true that the advanced features of Haskell can leave merely mortal programmers scratching their heads, but tasteful and judicious use of Haskell's features can yield inspiring results. For example, take a look at the very readable and expressive approach to implementing a Sudoku solver in Haskell described by Richard Bird in his 2006 paper and in his books Pearls of Functional Algorithm Design *(2010) and* Thinking Functionally with Haskell *(2014). I highly recommend either book!*

So why would I share an article that shows me failing at a task? Well, I've always believed that we can learn at least as much, if not more, from our failures

than from our successes! And maybe someone trying to follow the same path I walked along in 2006 might find this record interesting.

Day 1

I've been working on some tools to help me solve Sudoku puzzles in Scheme. That goal is actually secondary to the goal of learning to write better, more idiomatic Scheme programs, and so progress towards the Sudoku solver is frequently interrupted while I backtrack in an attempt to reorganize and optimize the supporting code.

I'm working on this both in a bottom-up and top-down direction. Bottom up, to develop something of a domain-specific Sudoku-solving language. Top down, because my higher-level strategy is not that clear at the outset, and is undergoing stepwise refinement.

My approach to solving Sudoku by computer is not to recursively explore every the tree of moves until a solution is found. That approach is easily doable but less interesting to me than trying to emulate some of the solving strategies that might be used by a human solver. Ultimately I would like the tool to assist a human solver, not replace her.

My tool of choice is PLT Scheme version 3.00. *Note: PLT Scheme is now known as Racket. This code should work with #lang r5rs.* For reasons to painful to go into just now, I have currently given up on using either SRFIs or SLIB for libraries and am writing everything in plain RSR5 Scheme. More on that later.

Day 2

Here is how I implemented some of my array primitives. I'll start in the middle:

```
(define map-board-row-elts
  (array2D-make-row-elt-mapper board-x-dim))
```

When treating a vector like a two-dimensional array, assuming that you store it in row-major order, the key piece of information needed to access elements is the size of the row. (More generally, for an n-dimensional array, to access an element you need to know the size of dimensions 1 to n - 1). On the top level of my program, the dimensions of my Sudoku board are known. But the Scheme array primitives don't know the dimensions. Rather than pass this information to every array access call, I create a function. The function **array2D-make-row-elt-mapper** captures the row size using closure and returns another function:

```
(define array2D-make-row-elt-mapper
  (lambda (x-dim)
    (lambda (array2D y-idx elt-handler)
      (array2D-map-row-elts array2D y-idx x-dim elt-handler))))
```

Note the nested lambda expressions. The top one receives the **x-dim** parameter, and the body of this function is another lambda expression. This second lambda expression is the return value of this function: it returns a function. This returned function invokes another function, **array2D-map-row-elts**, which we haven't defined yet, which requires an **x-dim** parameter.

In this case, the purpose of returning this function is to provide the functionality of another function, **array2D-map-row-elts**, but without the need to provide the **x-dim** parameter. The new function is a *curried function*. What is a curried function? Currying means taking a function with n parameters and creating a new function with n - 1 parameters. The new function provides a default value for the missing parameter, giving you a simplified API to the original function.

Common Lisp and other languages provide **curry** and **rcurry** functions which peel off parameters from the left or right of the paramter list, respectively. So our currying may not meet a strict definition, although we are doing the same thing: providing an interface function that will supply a value for one of the original parameters. Currying often uses a constant for these default parameters; in our case, we're using a variable, but the idea is similar. Scheme does not have a standard **curry** function, but it is easy enough to express currying by using nested lambda expressions. In Haskell, functions are curried automatically if you invoke them with a partial parameter list, which is confusing to beginners but immensely useful.

Again, **x-dim** is the curried parameter. Inside this lambda expression is a reference to **x-dim**. Scheme supports *lexical closure*; at the time this lambda expression is evaluated, the result is a function object that "closes over" or "captures" — that is, retains a reference to — the variable binding in effect when the lambda expression is evaluated.

This is the variable binding owned by the caller, the variable bound to the name **board-x-dim** in the calling function. It is *not a copy*: if some other piece of code updated the value in that variable, this function would see a different value for **x-dim** the next time it executes. This creates complex state with multiple possible "listeners." This is one of the reasons that Scheme programmers tend to prefer a functional style where variables are not altered after their initial binding.

If you are a C programmer, you may find this very confusing. You can think of it as if the returned function retains a pointer to the **x-dim** sent by the caller. The C runtime model is quite different, though: it has distinct lifetimes for variables. Local non-static ("automatic") variables that exist on the stack may be recycled when a given function returns, so it is unsafe to retain pointers to them (in fact, retaining a pointer to an automatic variable is a common beginner's bug in C). But in Scheme, bindings exist as long as there are references to them; when there are no more references to them, they may be garbage-collected (or not).

Let's look at the definition of the next function we call, **array2D-map-row-elts**:

```
(define array2D-map-row-elts
  (lambda (array2D y-idx x-dim elt-handler)
    (array2D-map-subrow-elts
      array2D 0 (- x-dim 1) y-idx x-dim elt-handler)))
```

In this function, we call yet another function: **array2D-map-subrow-elts**, which is more general (it will map any contiguous subset of the array elements). It is the general case, and **array2D-map-row-elts** is the more specific case (it maps the whole row), which means we can quite naturally define this specific case by parameterizing the general case. It isn't quite the same as currying, but it is a similar idea; we provide some fixed parameters representing the start of the row (index 0) and end (index x-dim - 1). Here is the general function:

```
(define array2D-map-subrow-elts
  (lambda (array2D x-first-idx x-last-idx y-idx x-dim elt-handler)
    (array2D-map-subrow-coords x-first-idx x-last-idx y-idx x-dim
                               (lambda (x-idx y-idx)
                                 (elt-handler (array2D-get array2D x-idx y-idx x-dim))))
```

Are we there yet?

This function receives all the parameters and walks through the array, starting with an arbitrary **x-idx** (in this case, zero), and proceeding until we have processed the last legal x index (in this case, **x-idx** - 1).

Note that instead of iteration to walk the array, I use *recursion*. I won't go into a full explanation of recursion here; suffice it to say that although many people feel that recursion is unnatural and does not reflect how they think about iteration in the "real world," after expressing algorithms recursively for a while, it starts to seem much more natural, and the proposition that recursion and iteration can be transformed into each other does not seem so strange.

This function is formulated using tail recursion, where the bindings are not referenced after the recursive function call. In C or Pascal, the compilers are generally not smart about this sort of thing, and recursing on a large array could result in a very deep set of stack frames. A language like Scheme is able to recognize that since the values that exist in previous function calls are not used, it is free to lose track of the previous values, so no deep stack is actually required. Since this is the case, the decision to use recursion was largely arbitrary. I chose it because I wanted to practice writing recursive functions.

The **letrec** may be unfamiliar. If the innermost function was not recursive, we could just call it like this, without ever binding the function to a name at all:

```
((lambda (x-idx y-idx)
   ...)
  0 y-idx)
```

In this case the innermost function is never bound to a name, because nothing has to refer to it by name. The expression around it would simply call it. This

is also known as *applying* parameters to the *anonymous* function object.

You might think that we could just do this with **let**. The semantics of **let** allow the call from outside the function, but the binding to the name **iter-x** would exist only *within* the body of the **let** expression. This body consists of the expression or expressions that occur after the binding list in the **let** expression. Since the definition of the function occurs as part of the binding list, not part of the body of the **let** expression, **let** will not work; **letrec** is designed specifically to allow definition of recursive or mutually recursive functions within the binding list.

So *now* are we there yet?

Almost. My final comment on this function concerns the use of the **elt-handler** parameter. The generated function allows the caller to pass in a handler function. Applying functions to functions is known as *higher-order programming*; support for higher-order programming is one of the essential features of Lisp and Scheme.

It might seem like this is much too complicated. Functions generating functions calling functions calling functions? Well, yes, this is a bit of overkill to access an array; normally we would use some standard functions for handling data structures. But the general style — creating a lot of small functions — appears very commonly in Scheme programs, and now you have had a taste of how a typical data structure library might be implemented.

I don't claim to be very experienced with Scheme yet, but I am happy to have gotten to the point where writing functions like **array2D-make-row-elt-mapper** and its supporting functions comes rapidly and naturally. Reading examples helps, but the lessons don't really click until you tackle a programming problem, and feel your mind begin to undergo rewiring. The level of abstraction, and thus the leverage you can gain from Scheme, become much higher than this, so stay tuned.

Day 3

OK, so I've given you a small view of one part of the array2D library and how it is implemented. Now let's look at my Sudoku solver from the top. Note that this is a hobby project, done in the rare scraps of free time I am able to piece together these days, by a relative novice in Scheme. In other words, it is not finished, polished, textbook work, so be kind.

Let's review what Sudoku is and consider some of the most elementary solving strategies. First, terminology. These are the terms I use; some are borrowed and some are my own. Other Sudoku geeks may use alternate terminology.

Box: the smallest unit; a single space for a number. I identify boxes using zero-based x, y coordinate pairs.

Open box: a box that does not have a number in it yet.

Coordinates: on a 9x9 grid, which does not represent the only possibility in Sudoku, the top row goes from (0, 0) to (8, 0) and the rightmost colum goes from (8, 0) to (8, 8).

Given: one of the numbers already provided in the puzzle.

Grid: in a 9x9 Sudoku puzzle, one of the 9 3x3 sub-parts; on a 16x16 puzzle, one of the 16 4x4 sub-parts, etc. Grids are not necessarily square; some published puzzles could contain 4x3 grids, for example.

Virtual row: a single group that must contain the set of possible numbers (in a 9x9 puzzle, 1 through 9). Note that each box is actually part of three virtual rows: its row, column, and grid.

Number set: for a 9x9 puzzle, the integers 1 through 9. I have seen Sudoku boards that allow the numbers 1 through 12, 1 through 16, 1 through 25, and 1 through 49. Sometimes letters are used: for example, the numbers 1 through 16 can be encoded using the hexadecimal digits 0..F (or 1..G), and the numbers 1 through 25 can be encoded by the letters A..Y. There are more exotic Sudoku involving the letters that make a specific word, but this is really just an encoding issue and does not fundamentally change the rules.

Sudoku rule: there is probably a good formal expression of this, but informally speaking, the Sudoku rule says that each box in each virtual row must contain one of the numbers in the number set.

By implication, each box cannot contain a number that is not a member of the number set; it also cannot contain a duplicate of another number in the same virtual row; in a completed puzzle each virtual row must be filled, which means the whole puzzle must be filled.

If you find that a puzzle you are working on seems to break this rule, either the puzzle was not valid to begin with, or you have made a solving error. The second case is more likely, but I have found errors in published puzzles.

There are several possible types of errors. The givens may violate the Sudoku rule, which will lead to an insoluble puzzle, or more than one solution may be possible, which could lead you to find a solution that follows the Sudoku rule but which does not match the published solution.

Candidate: one of the possible values that could legally be written in a box.

On an empty 9x9 grid, each box has as its candidates all the numbers 1 through 9, so no progress can be made towards a solution.

When the givens are taken into account, the set of candidates in the remaining open boxes in the neighborhood must be reduced to maintain the Sudoku rule. As the puzzle is filled in, the candidates are reduced further.

As you perform logical deductions on the board, you will remove candidates; when only one candidate remains for a box, the number should be written down and the candidates for boxes elsewhere in the neighborhood must have that

number removed; repeat until no open boxes remain. (This is actually in a nutshell the algorithm for solving any Sudoku puzzle, and translates directly into the main loop of a computer solution that uses human solving techniques rather than testing every possibile chain of possibilities).

Non-candidate: instead of writing down candidates as you work on the puzzle, you could also write down non-candidates: that is, numbers that have been ruled out for each box. When the set of non-candidates for a given box leaves only one remaining candidate in the number set, you can write down that possibility, and you must then add this number this to the sets of non-candidates elsewhere in the neighborhood. You could represent non-candidates by writing them down directly, although this can be confusing, or writing down the number set in each open box and crossing out numbers as they become non-candidates.

Single Candidate: the case when one candidate remains in a box; once you recognize this case, you should write in the number; the box is solved, you should then make sure to reduce the candidates in the neighborhood. Which brings us to another term.

Neighborhood: the combined row, column, and grid that contains a given open box. Neighborhood is a distinct concept from virtual row because if a number appears anywhere in the neighborhood containing a given box, it cannot be a valid candidate for the box; this is because the Sudoku rule applies to each box three ways.

Basic Solving Strategies

For demonstration purposes, we will use the following 9x9 puzzle (taken from *Sudoku: Easy to Hard Presented by Will Shortz Volume 3*):

```
-------------------------
| 9 . 8 | 3 . . | 7 4 2 |
| . 4 . | 5 8 . | . 9 . |
| 2 . . | . 7 . | 1 . 8 |
-------------------------
| . . 6 | 2 9 4 | 8 3 . |
| 4 8 . | . . . | . . 1 |
| 3 7 . | 6 . 8 | . 2 . |
-------------------------
| . 3 . | . 5 6 | . . . |
| 6 . . | 1 . . | . 8 3 |
| . 2 . | . . 3 | . 1 5 |
-------------------------
```

Let's look at how we might represent this board in Scheme, and then start coming up with functions to operate on it.

To allow my solver to be configured for different board sizes and grid dimensions, we'll provide bindings for the independent dimension values:

```
(define max-box-value 9)
(define board-x-dim 9)
(define board-y-dim 9)
(define grid-x-dim 3)
(define grid-y-dim 3)
```

To start solving our sample puzzle, let's use the following algorithm:

First, make a given board. A given board is a board array that contains only the givens. Here are the givens, as a list of list, where each list contains a coordinate pair as a list and a value:

```
(define given-list
  '(((4 0) 6) ((5 0) 5) ((7 0) 9) ((8 0) 1)
    ((6 1) 5)
    ((0 2) 4) ((1 2) 5) ((2 2) 9) ((3 2) 1) ((7 2) 7)
    ((4 3) 9) ((7 3) 3)
    ((2 4) 1) ((3 4) 8) ((5 4) 6) ((6 4) 7)
    ((1 5) 7) ((4 5) 4)
    ((1 6) 9) ((5 6) 4) ((6 6) 6) ((7 6) 2) ((8 6) 5)
    ((2 7) 4)
    ((0 8) 8) ((1 8) 2) ((3 8) 7) ((4 8) 5)))
```

Here's a function which will make a given board:

```
(define make-given-board
  (lambda (given-list)
    (let ((given-board
           (make-vector (* board-x-dim board-y-dim) #f))
          (setter! (array2D-make-setter board-x-dim)))
      (letrec ((do-write-givens
                (lambda (puzzle-array givens)
                  (if (not (null? givens))
                      (begin
                        (do-write-givens
                         puzzle-array (cdr givens))
                        (setter! puzzle-array
                                 (caaar givens)
                                 (cadaar givens)
                                 (cadar givens)))))))
        (do-write-givens given-board given-list))
      given-board)))
```

This is a pretty simple function, but let's look at a couple of points. First, the default value of our vector is **#f**, the canonical "false" value. This has no real purpose other than allowing us to use the elements of this array act as an argument to Scheme's **if** or **cond** and give the expected results. We've already seen how to use **letrec** to create a self-recursive helper function; we already know how the curried **setter!** function works. The body of this function is a

pretty straightforward example of recursion over a list. The arguments we pass to **setter!** are a little more interesting:

```
(setter! puzzle-array
         (caaar givens)
         (cadaar givens)
         (cadar givens))
```

These primitives provide shortcuts for expressing combinations of **car** and **cdr**, and are read inside out from right to left; **caaar** becomes **(car (car (car ...)))**, cadaar becomes **(car (cdr (car (car ...))))**.

An aside: I have a love-hate relationship with these constructions. On the one hand, they allow you to do quick deconstruction of arbitrary list structures, and that's a good thing because lists are very quick and dirty and you can easily put together data structures using lists. On the other hand, they allow you to do quick deconstruction of arbitrary list structures, and that's bad because they are not self-documenting in any way and nothing in them is named. Using something like a *property list* or record type is probably a better idea, so I will consider that later; Common Lisp provides *destructuring*, which would also be a valid option because it is self-documenting.

Anyway, that will build a given board. Now let's build a candidate board. The initial candidate list can be easily generated using another function:

```
(define make-initial-cand-list
  (lambda ()
    ((lambda (max)
       (letrec ((make-cand-list
                 (lambda (count)
                   (if (> count max) (list)
                       (cons count
                             (make-cand-list (+ count 1)))))))
         (make-cand-list 1)))
     max-box-value)))
```

I'm going to wrap this up in the form of a value generator. A value generator is a function that receives x and y indices and returns a new value for the board. In this case our value generator will ignore the parameters:

```
(define initial-cand-value-generator
  (lambda (x-idx y-idx)
    (make-initial-cand-list)))
```

Now we can use our **set-board-elts** primitive, which is a curried array-mapping function, to fill in the candidates. Each box gets a list containing the full set of candidates:

```
(define make-initial-cand-board
  (lambda ()
```

```
(let ((cand-board
       (make-vector (* board-x-dim board-y-dim) '())))
  (set-board-elts
   cand-board initial-cand-value-generator)
  cand-board)))
```

Now we've got our given board and our candidate board, and we can start smashing them together to shake out candidates, and begin finding values to go in our boxes. More on that next time.

Day 4

Last time, I wrote:

> Now we've got our given board and our candidate board, and we can start smashing them together to shake out candidates, and begin finding values to go in our boxes.

So let's do that. But first, I need to confess something. You may have found yourself wondering "You already have a data structure that contains the givens and their coordinates. Why make the **given-board** vector at all?" If you wondered that, congratulations. You're paying attention, possibly more than I am, and working towards writing code that is simple.

So what's the answer? The answer is "I don't have a good reason; the code just grew that way, and has not been optimally cleaned up yet." As I write this code, I'm learning to use Scheme more effectively. It has been an iterative process. Right now I'm in the part of the cycle that is excited to have the code successfully working on the first part of the Sudoku board solution. At some point I'll probably be in the part of the cycle that wants to rip up the code and make it smaller and simpler. Scheme lends itself well to stepwise refinement. This is both a blessing and a curse. A blessing, because it is easy to rewrite your code. In C++, assuming your design is not visibly horrible and doesn't crash, it is generally too painful to redo all those interfaces and class definitions and so working code often doesn't get improved unless there is a burning need. Scheme, especially if it is written in an appropriate functional style, is easy to improve locally. This also means that you can always improve it some more, and fall into the trap of improving it endlessly without solving an actual problem. That's the curse.

OK, now let's get to the meaty part, keeping in mind that this is code in progress. Here's how we map the givens and use them to cancel the first round of candidates. From the top:

```
(cancel-givens
 (make-given-board given-list)
 (make-initial-cand-board))
```

Simple enough so far, right? Now we just have to define **cancel-givens**, and we're done.

```
(define cancel-givens
  (lambda (given-board cand-board)
    (map-board-coords-elts
     given-board
     (lambda (x-idx y-idx elt)
       (if elt
           (cancel-given cand-board x-idx y-idx elt))))
    cand-board))
```

We know roughly what that map function is going to look like; it will call a function with the coordinates and elements of the given board. Now we just need to define **cancel-given**, and we're done. here's **cancel-given**. This one is big (most likely it could and should be further simplified):

```
(define cancel-given
  (lambda (cand-board x-idx y-idx given)
    (reduce-cand-list-to-given!
     cand-board x-idx y-idx given)
    (letrec ((filtered-neighborhood-unique-coords-list
              (make-filtered-coord-list
               (get-unique-neighborhood-coords
                x-idx y-idx) (list x-idx y-idx)))
             (cand-remover
              (make-cand-remover given))
             (coords-list-handler
              (lambda (coords-list)
                (if (not (null? coords-list))
                    (begin
                      (cand-remover
                       cand-board
                       (caar coords-list)
                       (cadar coords-list))
                      (coords-list-handler
                       (cdr coords-list)))))))
      (coords-list-handler
       filtered-neighborhood-unique-coords-list))
    cand-board))
```

Let's look at this one in a little more detail. Remember that this is called with each given. The first thing we do is turn the candidate list in the box with the given to a list containing only the given. Maybe that isn't the best representation, but it will do for now:

```
(define reduce-cand-list-to-given!
  (lambda (cand-board x-idx y-idx given)
```

```
    (set-board-elt!
     cand-board
     x-idx y-idx
     (list given))))
```

Note that this is not strictly functional in style. I have tried to move over to a functional style, but my roots are still in imperative programming, so a fully functional, referentially-transparent implementation does not come naturally quite yet. In general, I am at a bit of a loss for good idioms to handle the gamut of solving strategies on the board over time without introducing mutation, but that is probably from lack of experience in purely functional programming techniques and data structures. I could keep regenerating the whole board, but that seems strange to me. I may explore a stricter functional design in a future version.

Meanwile, let's continue on and look at the next task, which is to remove the given from the entire neighborhood. Recall that the neighborhood of a box is the combined row, column and grid. We want a list of coordinates. Simply combining the coordinates for the row, column and grid will generate duplicate sets of coordinates; we also don't want to process the box itself, or we'll remove the given we just set. So we first use **get-unique-neighborhood-coords**:

```
(define get-unique-neighborhood-coords
  (lambda (x-idx y-idx)
    (let ((grid-bounds (map-coords-to-grid-bounds x-idx y-idx))
          (acc (make-unique-coords-accumulator)))
      (map-board-row-coords y-idx (car acc))
      (map-board-col-coords x-idx (car acc))
      (map-board-subarray-coords
       (car grid-bounds)
       (cadr grid-bounds)
       (caddr grid-bounds)
       (cadddr grid-bounds)
       (car acc))
      ((cadr acc)))))
```

The function **map-coords-to-grid-bounds** uses a little bit of Scheme math to determine the bounds of the 3x3 grid our box is in. Here's how I do that:

```
(define map-coords-to-grid-bounds
  (lambda (x-idx y-idx)
    (list (get-grid-lower-bound x-idx grid-x-dim)
          (get-grid-upper-bound x-idx grid-x-dim)
          (get-grid-lower-bound y-idx grid-y-dim)
          (get-grid-upper-bound y-idx grid-y-dim))))
```

And here are the helper functions:

```
(define get-grid-lower-bound
```

```
  (lambda (idx dim)
    (* dim (floor (/ idx dim)))))

(define get-grid-upper-bound
  (lambda (idx dim)
    (- (* dim (ceiling (/ (+ 1 idx) dim))) 1)))
```

The particulars of Scheme math require some care if you want to make sure you get a whole number result rounded in the direction that you want. If you're used to using languages that don't do exact fractions, this may seem like too much work. This way seems to work, though, so we will move on for now.

Next, we use an accumulator (created by **make-unique-coord-accumulator**) to collect up all the coordinates. This isn't a numeric accumulator, but a list accumulator: when we create it, it holds a reference to an empty list, and as we call it with coordinates, the list is assembled. Our accumulator generator looks like this:

```
(define make-unique-coord-accumulator
  (lambda ()
    (let ((coords (list)))
      (list
       (lambda (x-idx y-idx)
         (if (not (member (list x-idx y-idx) coords))
             (set! coords
                   (cons (list x-idx y-idx) coords))))
       (lambda ()
         coords)))))
```

This is again not strictly functional; in a more functional style, we'd probably generate a new accumulator with a new list created from the previous one. But then the client that held the accumulator would have to mutate its binding in order to get the new accumulator, so it would have to be strictly functional as well – a sort of viral strict functionality would require rewriting all the upstream code. Something to think about!

This is yet another function generator, with a twist. We use a let to create a binding for a list, called **coords**. This is the list that will accumulate the coordinates. The actual return value of the function is… another list! This time, a list of two functions. The first function in the list receives coordinates and if the coordinate pair is not already present in the closed-over list, as determined by **member**, we destructively set coordinates to be the list with the new coordinate pair prepended. The second function in the list doesn't look like a function at all; in fact, it consists only of:

```
(lambda ()
  coords)
```

All that does is return the value held in the binding named **coords**. It's the

rough equivalent of a return statement in C, C++, or Java. The important thing to note here is that both of the functions in the list we return contain a reference to coordinates. The binding in question is the one that is in lexical scope when the lambda form is executed. Note that we could stuff these functions inside some other data structure; we could even put them in a single cons cell using set-car! and set-cdr! But for now I am still using the list as my "Swiss Army Knife" data structure, so we'll use that.

Returning to the innards of **get-unique-neighborhood-coords** let's look at what happens next:

```
(map-board-row-coords y-idx (car acc))
(map-board-col-coords x-idx (car acc))
(map-board-subarray-coords
 (car grid-bounds)
 (cadr grid-bounds)
 (caddr grid-bounds)
 (cadddr grid-bounds)
 (car acc))
```

I'm applying my accumulator to the row, the column, and the subarray. The map functions are unexceptional; they are wrappers around my 2D array implementation. To pass the four parameters that constitute the bounding rectangle of the subarray, I'm again using those all-purpose list functions **car**, **cadr**, **caddr**, and **cadddr**. These simply correspond to retrieving the first, second, third, and fourth element in a list.

The last expression in the function is the return value. Don't overlook its importance:

```
(cadr acc)
```

This actually calls the function contained in the second element of our accumulator list, which will retrieve the contents of the accumulator. Since it is the last expression in the function, **get-unique-neighborhood-coords** will return its value.

Now we're back in **cancel-given**. The next thing we do is call make-filtered-coord-list on our unique list. We do this to remove the box containing our given, so that we don't wind up removing that candidate as well:

```
(define make-filtered-coord-list
  (lambda (coords-list coords-pair)
    (if (null? coords-list) (list)
        (if (not (equal? coords-pair (car coords-list)))
            (cons (car coords-list)
                  (make-filtered-coord-list
                   (cdr coords-list) coords-pair))
            (make-filtered-coord-list
             (cdr coords-list) coords-pair)))))
```

This function is a textbook case of filtering out one member from a list, returning a new list. The key implementation detail here is that it recursively walks the list, building up a new list as it goes. As it stops to examine each element in the list, if it is not the equal to the item in question, it goes ahead and conses it on to the generated list; if it is, the element is not consed, and we just continue to recurse. You can see similar examples in *The Little Schemer*. Note that this function counts on the fact that **equal?** works properly on data structures that are not atoms and on atoms that are different objects in memory. In Scheme, Lisp, and other dynamic languages – basically, as soon as you get away from "unboxed" primitive data types – equality is a complex subject!

OK, are you still with me? We've got a list of the coordinates for the neighborhood we're examining. Next, we make a candidate remover, which is another function generator; when you pass it a candidate, it removes it from all the candidate list. That should be straightforward enough by now, so I won't belabor it. So, does it work?

Well, of course, it did not work on the first try. Or the second or third try! In fact, while I have presented this design top-down, I actually wrote it using a much more realistic mixture of top-down and bottom-up design. Sometimes the higher-order algorithm will be clear. This is often best worked out on paper. The **cancel-givens** and **cancel-given** function started out as a stub that called stubs. To test the implementation function-by-function, the stubs just returned fixed values. These were later fleshed out with real code. As I implemented more functions, I found that certain other function definitions seemed to naturally fall out, so wrote those next. Meanwhile, had already gone "bottom up" in the process of writing my 2D array functions, so my implementation hooked up in the middle. Then there was a bit of "impedance mismatch," where the functions I thought I would need turned out not to be precisely the functions I did need. I wound up iterating in both the "bottom up" and "top down" directions multiple times before getting even to this point. But using the DrScheme listener, turnaround on this kind of iteration is very quick.

The result of removing all the givens from the board, and crossing off candidates in the neighborhood of each given as I go, is a **cand-board** vector that looks like the one below. A list of only one value indicates that the value has been solved. I have marked the initial set of givens with square brackets instead of parentheses:

```
((2 3 7) (3 8) (2 3 7 8) (2 3 4) [6] [5] (2 3 4 8) [9] [1]
(1 2 3 6 7) (1 3 6 8) (2 3 6 7 8) (2 3 4 9) (2 3 7 8)
(2 3 7 8 9) [5] (4 6 8) (2 3 4 6 8)
[4] [5] [9] [1] (2 3 8) (2 3 8) (2 3 8) [7] (2 3 6 8)
(2 5 6) (4 6 8) (2 5 6 8) (2 5) [9] (1 2 7) (1 2 4 8)
[3] (2 4 6 8)
(2 3 5 9) (3 4) [1] [8] (2 3) [6] [7] (4 5) (2 4 9)
(2 3 5 6 9) [7] (2 3 5 6 8) (2 3 5) [4] (1 2 3) (1 2 8 9)
(1 5 6 8) (2 6 8 9)
```

```
(1 3 7) [9] (3 7) (3) (1 3 8) [4] [6] [2] [5]
(1 3 5 6 7) (1 3 6) [4] (2 3 6 9) (1 2 3 8) (1 2 3 8 9)
(1 3 8 9) (1 8) (3 7 8 9)
[8] [2] (3 6) [7] [5] (1 3 9) (1 3 4 9) (1 4) (3 4 9))
```

Note that there were 28 givens. The next important question to ask is "did we find any more solved boxes?" That is, did we reduce the candidate list in any of the non-given boxes to a length of 1, indicating only a single possibility in that box?

If you look closely, you will notice that, in fact, we have solved some boxes, simply by crossing out the givens and following through on the implications of those givens for the boxes in the neighborhood of each given. In the third-from-last row, which would be row 6 in our zero-based indexing, we see that there is a single 3 candidate in one of the candidate lists. This was not a given:

```
(1 3 7) [9] (3 7) (3) (1 3 8) [4] [6] [2] [5]
```

However, our code did not yet discover that we have made more progress. In the next installment, we'll look at how the code can make that discovery, and what the implications are. We'll also look how to apply some slightly less elementary solving rules to the board.

Day 5

Just a brief post today. I have been too busy to do much on my Sudoku solving program itself, but I did spend some time polishing up my array2D library. With revision this library has finally assumed what I feel is its "natural" shape. The optimal design was not clear at the outset, but with some use I feel I've discerned the implementation that minimizes the amount of redundant code and thus the number of places that possible errors may be hiding. I'm sure it could be even more concise; I have a feeling that a little macro magic could cut the lines of code in half, but the shape of that implementation has not yet become clear to me, and I have an allergy to using techniques that don't provide a clear advantage.

It is really quite a simple library, but in order to provide some assurance of correctness, I also wrote a series of PLT Scheme test cases. I have not figured out how to use DrScheme to tell me if this test suite covers all of the code, or if this is even possible, but it is at least nearly complete.

The library does not do meaningful error-checking. Scheme textbook code rarely does anything by way of error-checking. As a result I'm deficient on strategies I could use to address this, especially when trying to stick to R5RS Scheme.

It isn't possible to code around every conceivable error. For some applications, it makes sense to perform verification any time data is read from or written to memory. We can't operate at that level. We also shouldn't put in effort to detect flaws in the implementation itself. Given our partially functional approach, in which we don't update any top-level definitions, there aren't a lot of unexpected

states our library can get into. We should focus our attention on the response to incorrect inputs. There are only a couple of classes of errors that the library should expect:

1. The set of array boundary errors. Because coordinate pairs are mapped to a single index on a vector, many of possible boundary errors will result in an incorrect cell being set or get. Only cell references that exceed the internal representation will generate a meaningful run-time error, and this error will not tell us if the x or y index was out of bounds.

2. The set of errors that can arise from receiving an incorrect value-generator or mapping function. In Scheme, a function signature really consists only of the name and the number of parameters, and does not include type information about the parameters. Supplying an incorrect number of parameters will be caught with a meaningful error at run-time but because the passed functions may be called via several levels of indirection, and the function may not be bound to a named variable, it may be difficult to glean from the error message just what has gone wrong. (Another way to say this is that DrScheme has thrown away too much context information).

Type errors are potentially more insidious, but in my implementation these functions will tend to receive an array (vector), integer indices, and possibly another function. The result of doing a vector access on an integer or function, or treating a vector or function like an integer, should at least fail fast with a moderately meaningful error.

There are richer language constructs that could address these problems. For example, either Common Lisp or Dylan would allow me to specify type information for the parameters. The runtime is then capable of flagging type errors as soon as they happen, or better yet, verify type safety across functions and generate code that operates on POD (plain old data). Dylan also supports limited types, although implementation support in Gwydion's **d2c** is spotty. Limited types would allow range checking further upstream. Of course, if I wrote this program in Common Lisp or Dylan, I would not bother simulating an array. If our language supported exception handling, I could wrap the top-level operations to catch exceptions without too much modification inside all my functions.

Another approach to bounds-checking in my 2-D array would be to encapsulate the array data together with metadata on its x and y dimensions. I am doing this to a limited extent with the closure generation, but the captured x-dimension is not used to validate that the x-index is within bounds. Other array implementations use a record or a closure to wrap up the representation itself. But then it is more difficult to display the contents of the array, and if we are going to start down that road, we might as well try to generalize our library for an n-dimensional array. That seems like overkill, so for now we will call **array2D** complete as it is.

The Code (Updated March 2025)

Looking at the old source code, it is clear that this is a snapshot of my work in progress. Not every function seems to be used. Some of them I likely wrote "on spec," thinking about what the higher-level code *might* require. That's a good way to experiment and learn, which is all I really wanted from this effort. I had not gotten to the point where I finished the higher-level code that used this scaffolding to implement a variety of Sudoku-solving strategies.

So, the code is mostly scaffolding to support a top floor that I didn't complete. Many functions lack comments, and some have only minimalist comments. Ideally there would be a third stage, too, after completing the top-level code: a "now that it works, refactor everything to be as simple and clear as possible" stage.

Because of its obvious work-in-progress status, I have not put this code on GitHub, but I have made the files available for anyone who wants to peruse them, study the functions, laugh at my Scheme programming skills (clearly a work in progress), or whatever. Knock yourself out.

I used the latest version of DrRacket, formerly PLT Scheme, to run these files. I set the environment to use R5RS mode, supporting the fifth version of the Scheme language as described in the specification document here.

The first two files are not intended to be run by themselves, but contain support functions:

array2d.scm

array2d-board-support.scm

The "top level" code is in **sudoku-vec.scm**. It builds a Sudoku board and applies an initial "cancel givens" strategy to solve it, but the resulting board is not fully solved.

sudoku-vec.scm

There is a fourth file, an unfinished test suite. It looks like I was using this code to debug the support functions:

sudoku-vec-tests.scm

It did not run correctly, which leads me to believe that I had changed the support functions without changing the tests, or wrote the tests without ever finishing the process of running and testing the tests.

Fortunately, although my Scheme is rusty, I recall enough to be able to patch up the tests and get them to run. In the process I found and fixed a few obvious bugs in the support functions.

So, thank you, past me: you did not finish this Sudoku solver, but at least you left me some bread crumbs pointing to things that needed to be tested and fixed.

The unfinished test file appears to run correctly now, but of course this is no guarantee that all the code is bug-free. Use it for any purpose at your own risk!

In a perfect world, at least as I define it, I'd have enough free time to one day come back to, and finish, all my unfinished programming projects. In the imperfect world we actually have, though, I need to devote most of my attention to work that I can get paid for. Don't get me wrong — I often enjoy my paid work quite a bit, especially when I can learn new things, but it is not always what I would choose to do for my own learning and enjoyment.

Maybe if I can ever retire, I'll be able to pass this way again and play with Scheme some more. But in case I never do — I leave that work to you.
