Clarke, Garbary, Morland, Petrick, Roh, Watson

Assignment:	10
Due:	**Tuesday**, December 2, 2025 9:00 pm
Coverage:	L21
Language level:	Intermediate Student with `lambda`
Allowed recursion:	Any
Files to submit:	`sudoku.rkt`

Assignment policies:

- Make sure you read the official assignment post on **Piazza**.
- Unless otherwise indicated, you may use abstract list functions and/or `lambda` to answer any question.
- Unless otherwise indicated, you may define helper functions as needed.
- Helper functions you write that are used by only one function **should be encapsulated within a `local`**. Helper functions used by multiple functions can be defined globally, unless stated otherwise in the question. Functions written for testing purposes can be defined globally.
- Functions and symbols must be named exactly as they are written in the assignment questions.
- You test cases must provide full coverage, i.e., no highlighting, but **you do not need to provide a purpose or contract for any function**.

Place your solutions for this assignment in the file `sudoku.rkt`. There are **5** parts in total.

1. **[25%]**: We will be working with `(listof (listof X))`, all of the same length. We start with a data definition:

```
;; A (matrixof X) is a (listof (listof X))
;; Requires: all (listof X) have the same length.
```

To make our later code simpler, we are going to start by writing a few higher-order functions that work on such values.

(a) **[5%]**:

Write a predicate `all-satisfy?: (X -> Bool) (matrixof X) -> Bool`.

It determines if **every** value in the matrix satisfies the predicate.

(We say a value **satisfies** a predicate if the predicate produces `true` when applied to that value. For example, 3 satisfies `integer?` since `(integer? 3)` ⇒ `true`. But 3 does not satisfy `symbol?` because `(symbol? 3)` ⇒ `false`.)

For example,

```
(check-expect (all-satisfy? integer? '((2 3 4) (5 6 7)))
              true)
(check-expect (all-satisfy? integer? '((2 3 4) (5 six 7)))
              false)
```

(b) **[5%]**:

Also write a predicate `any-satisfy?: (X -> Bool) (matrixof X) -> Bool`.

It determines if **any** value in the matrix satisfies the predicate.

(If you understand De Morgan's Laws, you can write this function in one line using `all-satisfy?` as a helper. If not, it will closely resemble `all-satisfy?`.)

```
(check-expect (any-satisfy? symbol? '((2 3 4) (5 6 7)))
              false)
(check-expect (any-satisfy? symbol? '((2 3 4) (5 six 7)))
              true)
```

(c) **[15%]**:

Write a function `find-where: (X -> Bool) (matrixof X) -> (list Nat Nat)`.

It produces a list storing the column number and row number of the first value that satisfies the predicate.

Throughout this assignment, "first" means the location that is encountered earliest when reading left to right, and top to bottom. (That is, in the usual way for written English.)

Start counting from zero. That is, the leftmost column is column zero, and the topmost row is row zero.

For example,

```
(define wherematrix '(( 1     2     3     4     )
                      ( 4     5     (3 6) (1 2) )
                      ( (7)   8     9     ()    )))
(check-expect (find-where list? wherematrix)  '(2 1))
(check-expect (find-where empty? wherematrix) '(3 2))
(check-expect (find-where integer? wherematrix)  '(0 0))
```

If you wish, you may make this function have a `;; Requires:` clause that some value in the `Matrix` satisfies the predicate. **Every `Matrix` we test this function with will contain such a value.**

However, you might find it useful to make this function indicate that the `Matrix` contains no value that satisfies the predicate. Be cautious of how you indicate this.

Keep these functions in mind as you work on the rest of the assignment. Using them well will save a lot of effort.

In this assignment you will be working with Latin squares.
A square grid of size n has the *Latin square property* if:

(a) each row contains each of $1, 2, 3, \ldots, n$ exactly once, and

(b) each column contains each of $1, 2, 3, \ldots, n$ exactly once.

Such a grid is called a *Latin square*.
For example, on the right is a Latin square of size 4.

1	3	2	4
4	2	1	3
2	4	3	1
3	1	4	2

Latin squares are a part of the structure underlying various logic puzzles. There is an excellent Open-Source collection including Latin square puzzles. Some puzzles have been renamed to avoid infringing trademarks, but you might like to try playing: Keen (KenKen), Solo (Sudoku), Towers, or Unequal.

We are going to write a generic algorithm that can solve any Latin square puzzle, including Sudoku.

To solve a Latin square puzzle, we are given a grid with some values filled in, and we seek to fill in the rest of the grid, satisfying the Latin square property. In each cell, we will store the list of all the values (no duplicates) that could satisfy the Latin square property, as a list in **increasing** order. When a cell contains only one possibility, we replace the list with the `Nat` it contains. When every cell contains a `Nat`, we have found a Latin square.

We define:

```
;; A Cell is a (anyof (listof Nat) Nat)
;; Requires: Nat values are positive.
;; List values contain no duplicates, and are in increasing order.

;; A Puzzle is a (matrixof Cell)
```

```
;; A Single is a (list Nat).  That is, a list of length exactly 1.

;; A Solution is a (matrixof Nat).
```

The first thing we need to do is make it easy to encode a `Puzzle` in Racket.

2. **[5%]**: Write a function `strings->puzzle`. It consumes a `(listof Str)` containing only digits and question marks `"?"`, and produces a `Puzzle` where each digit has been replaced by a `(listof Nat)` containing only the corresponding `Nat`, and each question mark has been replaced by a `(listof Nat)` containing the positive integers from 1 to the size of the `Puzzle`. (In order to produce a valid `Puzzle`, all the strings must be of the same length, and each digit must be between 1 and the size of the puzzle.)

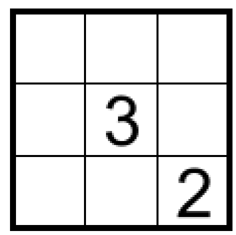
   For simplicity, we are restricting ourselves to puzzles containing only the numbers 1–9.

   (If you want to test your code with larger inputs, you could use Base 36, where we use ‘a’ for 10, ‘b’ for 11, ‘c’ for 12, and so on, up to ‘z’ for 35. We will **not** test your code with anything larger than 9.)

   > For this assignment, you may use the `char->integer` function. It consumes a UNICODE character and produces the corresponding number. For example, `(char->integer #\0)` ⇒ `48`, and `(char->integer #\9)` ⇒ `57`.

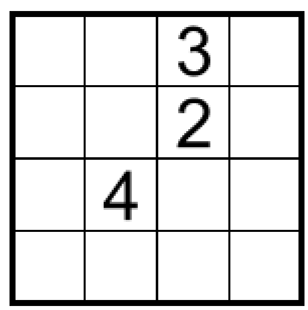
   For example, here is a test to translate the puzzle on the right into Racket code:

```
(check-expect (strings->puzzle '("???"
                                  "?3?"
                                  "??2"))
              '(( (1 2 3) (1 2 3) (1 2 3) )
                ( (1 2 3) (3)     (1 2 3) )
                ( (1 2 3) (1 2 3) (2)     )))
```



   And here is a test for a larger puzzle:

```
(check-expect (strings->puzzle '("??3?"
                                  "??2?"
                                  "?4??"
                                  "????"))
   '(( (1 2 3 4) (1 2 3 4) (3)       (1 2 3 4) )
     ( (1 2 3 4) (1 2 3 4) (2)       (1 2 3 4) )
     ( (1 2 3 4) (4)       (1 2 3 4) (1 2 3 4) )
     ( (1 2 3 4) (1 2 3 4) (1 2 3 4) (1 2 3 4) )))
```



3. **[25%]**: At a high level, the algorithm to solve a Latin square will be as follows:

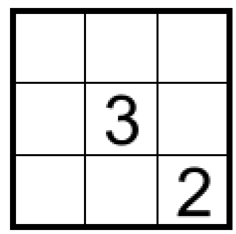
   (a) Choose a cell.

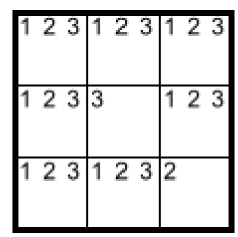
   (b) For the chosen cell, choose a value, and record that the other cells in the same column and row must not have the same value.

   (c) We now have a puzzle with fewer possibilities. Recursively solve it, until either it is fully solved, or we find a cell where no value is possible. (The latter is a *contradiction*, indicating that we made a wrong choice somewhere. We will discuss how to solve this later.)

The details of each choice will be specified as we move forward.

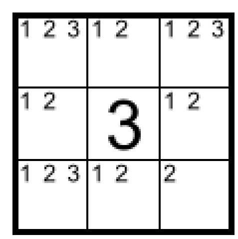
As an example, we will walk through the process of solving the following Latin square puzzle:





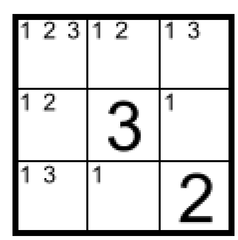
In each cell, we record a list of all the values that are possible. Since we know what some cells contain, each of those cells shall contain a list of only one item, which we call a `Single`. In Racket:

```
'(( (1 2 3)    (1 2 3)    (1 2 3) )
  ( (1 2 3)    (3)        (1 2 3) )
  ( (1 2 3)    (1 2 3)    (2)     ))
```



We choose the first cell that contains a `Single`: the middle cell, which must contain a 3. Fill in that possibility: replace the `'(3)` with `3`. From each cell in the same row and column, remove the 3. In Racket:

```
'(( (1 2 3)    (1 2)      (1 2 3) )
  ( (1 2)      3          (1 2)   )
  ( (1 2 3)    (1 2)      (2)     ))
```

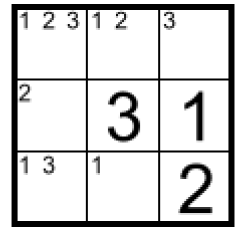


Again, choose the first cell that contains a `Single`: the bottom-right cell, which must contain a 2. Fill in that possibility: replace `'(2)` with `2`. From each cell in the same row and column, remove the 2.

```
'(( (1 2 3)    (1 2)      (1 3)   )
  ( (1 2)      3          (1)     )
  ( (1 3)      (1)        2       ))
```

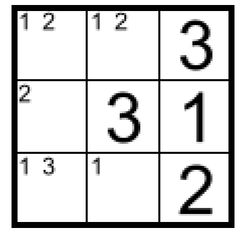
Now these updates have caused a new cell to contain a `Single`. Fill this in as before.

```
'(( (1 2 3)   (1 2)      (3)      )
  ( (2)       3          1        )
  ( (1 3)     (1)        2        ))
```



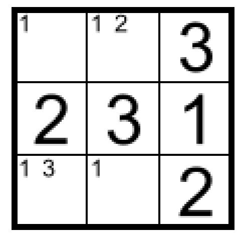
Continuing this process:

```
'(( (1 2)     (1 2)      3        )
  ( (2)       3          1        )
  ( (1 3)     (1)        2        ))
```



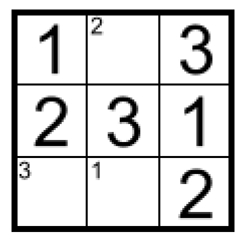
And so:

```
'(( (1)       (1 2)      3        )
  ( 2         3          1        )
  ( (1 3)     (1)        2        ))
```



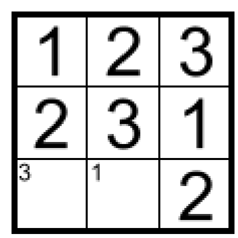
A little further:

```
'(( 1         (2)        3        )
  ( 2         3          1        )
  ( (3)       (1)        2        ))
```



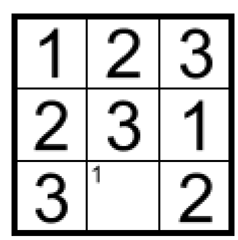
Almost there:

```
'(( 1         2          3        )
  ( 2         3          1        )
  ( (3)       (1)        2        ))
```



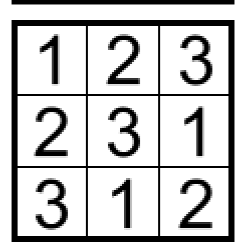
One more:

```
'(( 1         2          3        )
  ( 2         3          1        )
  ( 3         (1)        2        ))
```



And finally:

```
'(( 1         2          3        )
  ( 2         3          1        )
  ( 3         1          2        ))
```



Write a function `remove-singles`. It consumes a `Puzzle` which may contain one or more `Single` values, and produces `Puzzle` which does not contain any `Single` values, following the algorithm described above.

You might find it useful to write helper functions to:

- determine if a `Puzzle` contains at least one `Single`;
- determine the coordinates of the first `Single` in a `Puzzle`;
- get and set the value at a particular coordinate in a `Puzzle`;
- determine the value of the first `Single` in a `Puzzle`;
- remove a particular `Nat` from each list in a particular row of a `Puzzle`; likewise for a particular column.

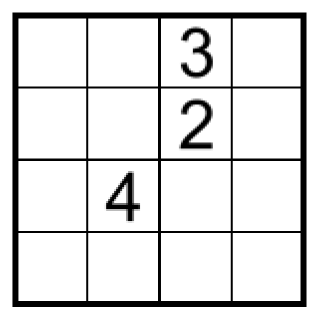
Consider carefully how the higher order functions you wrote earlier could help.

You may find that you need to write additional helper functions to make these helpers!

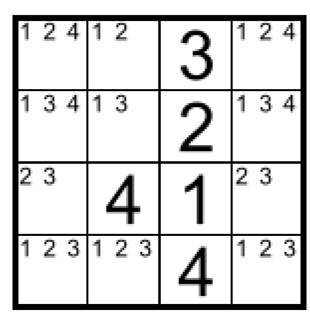
The example above corresponds to:

```
(check-expect (remove-singles (strings->puzzle '("???"
                                                 "?3?"
                                                 "??2")))
              '((1 2 3)
                (2 3 1)
                (3 1 2)))
```

But consider what `remove-singles` does with this size 4 puzzle:



After a few steps we fill in the 1 and the 4, and we end with this:
This is all that `remove-singles` is expected to do.



Here is the corresponding test:

```
(check-expect (remove-singles (strings->puzzle '("??3?"
                                                  "??2?"
                                                  "?4??"
                                                  "????")))
              '(( (1 2 4) (1 2)   3       (1 2 4) )
                ( (1 3 4) (1 3)   2       (1 3 4) )
                ( (2 3)   4       1       (2 3)   )
                ( (1 2 3) (1 2 3) 4       (1 2 3) )))
```

4. **[35%]**: The `remove-singles` function efficiently solves some Latin square puzzles, but it only partly solves others. Also, to solve a puzzle such as Sudoku, not any Latin square will suffice. It needs to be a Latin square that satisfies some additional constraints.

   To describe the additional constraints, we will use a predicate that consumes a fully-solved `Puzzle` (that is, a `(listof (listof Nat))`, all of the same length).

   Here are some predicates we will use in examples.

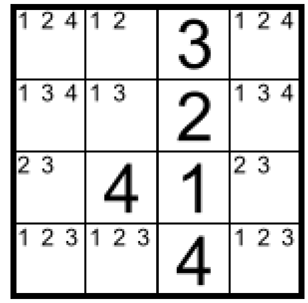
```
;; Ignore the parameter; always produce true.
;; yes: Any -> Bool
(define (yes x) true)

;; Ignore the parameter; always produce false.
;; no: Any -> Bool
(define (no x) false)

;; Determine if the diagonal of p has a 2 in it.
;; diagonal-has-2?: Solution -> Bool
(define (diagonal-has-2? p)
  (and (not (empty? p))
       (or (= 2 (first (first p)))
           (diagonal-has-2? (map rest (rest p))))))

(check-expect (diagonal-has-2? '((3 2 1)
                                 (2 1 3)
                                 (1 3 1))) false)
```

   Recall that `remove-singles` did not fully solve the puzzle we saw on page 7. But the cells are constrained; if there is a solution, the top-left corner must be 1, 2, or 4. So now we will try these possibilities, one after the other, until we find a solution that works.

   Remember: we always make a guess in the first cell that has a choice, reading left to right, and top to bottom (again, in the usual way for written English).

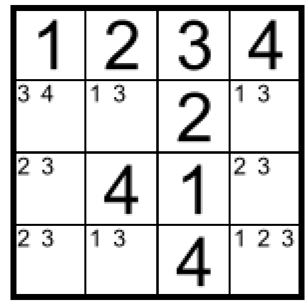
We start from:

```
'(( (1 2 4) (1 2)   3         (1 2 4) )
  ( (1 3 4) (1 3)   2         (1 3 4) )
  ( (2 3)   4       1         (2 3)   )
  ( (1 2 3) (1 2 3) 4         (1 2 3) ))
```



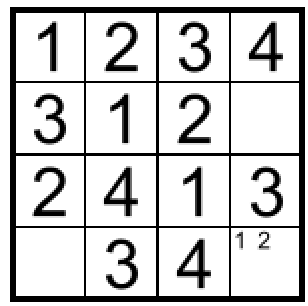
We make a guess, and we put `'(1)` in the top left corner.
Then we remove the singles, and we get:

```
'(( 1       2       3       4       )
  ( (3 4)   (1 3)   2       (1 3)   )
  ( (2 3)   4       1       (2 3)   )
  ( (2 3)   (1 3)   4       (1 2 3) ))
```



Now we see that the first `Cell` in the second row must be a 3 or a 4.
Let us guess 3. Then `remove-singles` should give us:

```
'(( 1       2       3       4       )
  ( 3       1       2       ()      )
  ( 2       4       1       3       )
  ( ()      3       4       (1 2)   ))
```

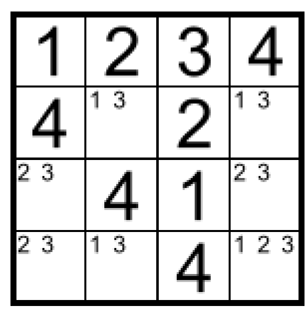


Oh dear! We have some cells where no value is possible. If we ever find a puzzle that is impossible to solve, we produce `empty`.

This means that when we made our last choice, we should try another choice. In the code where it guessed it was 3 in the second row, it has to try another guess.

Instead of putting a 3, try a 4.

```
'(( 1       2       3       4       )
  ( 4       (1 3)   2       (1 3)   )
  ( (2 3)   4       1       (2 3)   )
  ( (2 3)   (1 3)   4       (1 2 3) ))
```



This seems OK so far. So now we choose between 1 and 3. Suppose it is a 1. Now `remove-singles` finds a `Solution`:

```
'(( 1       2       3       4       )
  ( 4       1       2       3       )
  ( 3       4       1       2       )
  ( 2       3       4       1       ))
```

1	2	3	4
4	1	2	3
3	4	1	2
2	3	4	1

There is one thing left to check: our predicate.

We call the predicate on the `Solution`. If it produces `true`, produce the `Solution`. Otherwise, backtrack.

If we were using the predicate `yes`, this solution is acceptable.

But if we were using the predicate `diagonal-has-2?`, this solution fails. In the recursion, we step back to our last choice: instead of a 1 in the second cell of the second row, try a 3:

```
'(( 1       2       3       4       )
  ( 4       3       2       1       )
  ( (2 3)   4       1       (2 3)   )
  ( (2 3)   1       4       (2 3)   ))
```

1	2	3	4
4	3	2	1
2 3	4	1	2 3
2 3	1	4	2 3

And so we make one more guess, and find this solution. Since this solution satisfies the predicate, we produce it:

```
'(( 1       2       3       4       )
  ( 4       3       2       1       )
  ( 2       4       1       3       )
  ( 3       1       4       2       ))
```

1	2	3	4
4	3	2	1
2	4	1	3
3	1	4	2

Write `solve-latin` with the following contract:

```
;; solve-latin: (Solution -> Bool) Puzzle -> (anyof Solution empty)
```

If this `Puzzle` has one or more `Solution` that satisfies the predicate, `solve-latin` produces the first such solution. Otherwise, `solve-latin` produces `empty`.

For example,

```
(define 23puzzle (strings->puzzle '("???"
                                    "?3?"
                                    "??2")))

(check-expect (solve-latin yes 23puzzle)
              '((1 2 3)
                (2 3 1)
```

```
                (3 1 2)))

(define 324puzzle (strings->puzzle '("??3?"
                                     "??2?"
                                     "?4??"
                                     "????")))

(check-expect (solve-latin yes 324puzzle)
              '((1 2 3 4)
                (4 1 2 3)
                (3 4 1 2)
                (2 3 4 1)))

(check-expect (solve-latin diagonal-has-2? 324puzzle)
              '((1 2 3 4)
                (4 3 2 1)
                (2 4 1 3)
                (3 1 4 2)))

(check-expect (solve-latin no 324puzzle) empty)
```

5. **[10%]**: We said we would solve Sudoku, and we are almost there.

   A Latin square is a solution to Sudoku if it has size 9, and can be broken up into nine 3×3 sub-squares, each of which contains the numbers 1–9 exactly once.

   For example, below is a Latin square. But it is not a solution to a Sudoku, because of the nine sub-squares, only the one highlighted in blue contains each of the numbers 1–9 exactly once.

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

Here is a puzzle for which this is a solution:

```
(define sample-sudoku (strings->puzzle '("?7????1?8"
                                         "??5?1??8?"
                                         "?6?7?5???"
                                         "2??93????"
                                         "3?1???8?6"
                                         "????28??1"
                                         "???2?7?1?"
                                         "?1??4?2??"
                                         "8?2????6?")))
```

And we should get this solution using the predicate `yes`:

```
(check-expect
 (solve-latin yes sample-sudoku)
 '((4 7 3 5 6 2 1 9 8)
   (6 2 5 3 1 4 7 8 9)
   (1 6 4 7 8 5 9 3 2)
   (2 4 8 9 3 1 6 7 5)
   (3 5 1 4 7 9 8 2 6)
   (7 3 9 6 2 8 5 4 1)
   (5 8 6 2 9 7 3 1 4)
   (9 1 7 8 4 6 2 5 3)
   (8 9 2 1 5 3 4 6 7)))
```

Write a function `sudoku?` with the following contract:

```
;; sudoku?: Solution -> Bool
```

It consumes a `Solution`, which shall be square and of size 9, and determines if it satisfies the additional Sudoku restriction.

It should **not** also check that what it consumes is a Latin square, and it does **not** need to verify that it is of size 9.

For example, this is not a Latin square, but it **does** satisfy the additional Sudoku restriction:

```
(check-expect (sudoku? '((1 2 3 4 5 6 7 8 9)
                         (4 6 5 7 8 9 1 2 3)
                         (7 8 9 1 2 3 4 5 6)
                         (4 5 6 7 8 9 1 2 3)
                         (1 2 3 4 5 6 7 8 9)
                         (7 8 9 1 2 3 4 5 6)
                         (7 8 9 1 2 3 6 5 4)
                         (4 5 6 7 8 9 1 2 3)
                         (3 2 1 4 5 6 7 8 9))) true)
```

This one fails the additional Sudoku restriction, in the bottom-right sub-square:

```
(check-expect (sudoku? '((1 2 3 4 5 6 7 8 9)
                         (4 5 6 7 8 9 1 2 3)
                         (7 8 9 1 2 3 4 5 6)
                         (4 5 6 7 8 9 1 2 3)
                         (1 2 3 4 5 6 7 8 9)
                         (7 8 9 1 2 3 4 5 6)
                         (7 8 9 1 2 3 4 5 7)
                         (4 5 6 7 8 9 1 2 3)
                         (1 2 3 4 5 6 7 8 9))) false)
```

We can now solve Sudoku puzzles, or create additional tests.

For example, to solve this puzzle, we can run the code:

```
(solve-latin sudoku? (strings->puzzle '("48?9?2???"
                                        "?93??5?86"
                                        "6??3???1?"
                                        "???5?1?9?"
                                        "????9????"
                                        "?3?2?6???"
                                        "?7????3??5"
                                        "24?1??36?"
                                        "???6?4?27")))
```

This takes about 2.8 seconds on my machine. (Different puzzles will be slower or faster. This is a fairly fast one.)

This concludes the list of questions for you to submit solutions. Don’t forget to always check the basic test results after making a submission.