Exercise 2.42.

The “eight-queens puzzle” asks how to place eight queens on a chessboard so that no queen is in check from any other (i.e., no two queens are in the same row, column, or diagonal). One possible solution is shown in figure 1. One way to solve the puzzle is to work across the board, placing a queen in each column. Once we have placed $k-1$ queens, we must place the kth queen in a position where it does not check any of the queens already on the board. We can formulate this approach recursively: Assume that we have already generated the sequence of all possible ways to place $k-1$ queens in the first $k-1$ columns of the board. For each of these ways, generate an extended set of positions by placing a queen in each row of the kth column. Now filter these, keeping only the positions for which the queen in the kth column is safe with respect to the other queens. This produces the sequence of all ways to place k queens in the first k columns. By continuing this process, we will produce not only one solution, but all solutions to the puzzle.

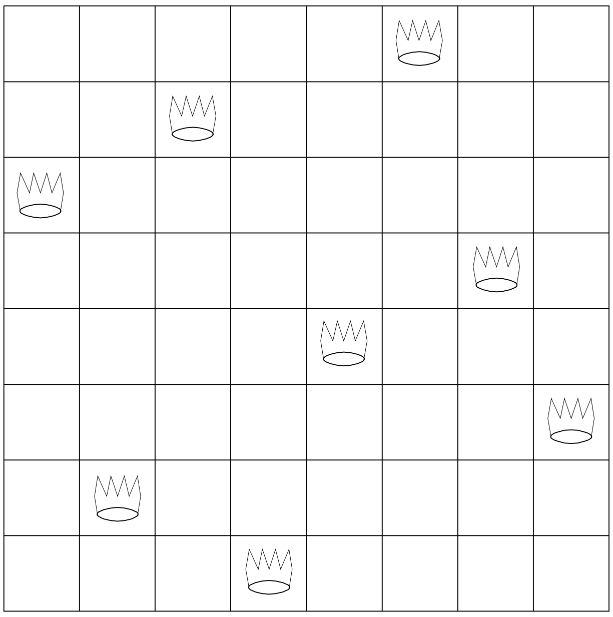

Figure 1. A solution to the eight-queens puzzle.

We implement this solution as a procedure `queens`, which returns a sequence of all solutions to the problem of placing n queens on an $n \times n$ chessboard. `Queens` has an internal procedure `queen-cols` that returns the sequence of all ways to place queens in the first k columns of the board.

```
(define (queens board-size)
  (define (queen-cols k)
    (if (= k 0)
        (list empty-board)
        (filter
         (lambda (positions) (safe? k positions))
         (flatmap
          (lambda (rest-of-queens)
            (map (lambda (new-row)
                   (adjoin-position new-row k rest-of-queens))
                 (enumerate-interval 1 board-size)))
          (queen-cols (- k 1))))))
  (queen-cols board-size))
```

In this procedure `rest-of-queens` is a way to place $k-1$ queens in the first $k-1$ columns, and `new-row` is a proposed row in which to place the queen for the kth column. Complete the program by implementing the representation for sets of board positions, including the procedure `adjoin-position`, which adjoins a new row-column position to a set of positions, and `empty-board`, which represents an empty set of positions. You must also write the procedure `safe?`, which determines for a set of positions, whether the queen in the kth column is safe with respect to the others. (Note that we need only check whether the new queen is safe – the other queens are already guaranteed safe with respect to each other.)

Answer.

Before setting out to represent sets of board positions, we have to clarify two crucial terminologies: *position* and *positions*. The term *position* stands for the location of a queen, whereas *positions* indicates a composition of the chess game, that is, a way of distributing all the queens on the board.

We usually refer to a piece on the chess board by the column and row it takes up. This indicates that we should represent sets of board positions using pairs[1]:

```
(define (make-pos col row)
  (list col row))

(define (col pos)
  (car pos))

(define (row pos)
  (car (cdr pos)))
```

Using this representation, `adjoin-position` can be implemented by `append`ing the existing positions by the position of a new queen:

```
(define (adjoin-position row col positions)
  (append positions (list (make-pos col row))))
```

An empty set of positions is simply denoetd by the empty list:

```
(define empty-board nil)
```

The queen in the kth column is said to be in safe with respect to the rest $k-1$ queens when

- the queen in the kth column is not threatened by the first one among the rest $k-1$ queens, and,
- the queen in the kth column is safe with respect to the rest $k-2$ queens.

Thus, we can recursively reduce the problem of placing a given number of queens to the problem of placing fewer number of queens of smaller chessboard. Consider this reduction rule carefully, and convince yourself that we can use it to describe an algothrim if we specify the following degenerate case:

- If the queen in the kth column is the only one on the chessboard, we should identify that the queen is safe.

We can easily translate this description into a recursive procedure:

```
(define (safe? new-col positions)
  (let ((new-queen (car (filter (lambda (pos)
                                  (= (col pos) new-col))
                                positions)))
        (rest-queens (filter (lambda (pos)
                               (not (= (col pos) new-col)))
                             positions)))
    (cond ((null? rest-queens) #t)
          ((threatens? new-queen (car rest-queen)) #f)
          (else (safe? new-col (cdr positions))))))
```

1. Again, here we also represent a pair as a list of two elements, the practice we adopted in implementing `prime-sum-pair?` above.

```
(define (threatens? q1 q2)
  (or (= (row q1) (row q2))
      (= (abs (- (col q1) (col q2)))
         (abs (- (row q1) (row q2))))))
```