```
;;;;   __    __         _
;;;;   \ \  / /__ _ __ (_) ___ ___
;;;;    \ \/ / _ \ '_ \| |/ __/ _ \
;;;;     \  /  __/ | | | | (_|  __/
;;;;      \/ \___|_| |_|_|\___\___|
;;;;
;;;;
;;;; Copyright 2017-2024 Venice
;;;;
;;;; Licensed under the Apache License, Version 2.0 (the "License");
;;;; you may not use this file except in compliance with the License.
;;;; You may obtain a copy of the License at
;;;;
;;;;     http://www.apache.org/licenses/LICENSE-2.0
;;;;
;;;; Unless required by applicable law or agreed to in writing, software
;;;; distributed under the License is distributed on an "AS IS" BASIS,
;;;; WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
;;;; See the License for the specific language governing permissions and
;;;; limitations under the License.

;;;; Sudoku solver

;;;; Constraints for a 9x9 Sudoku
;;;;
;;;; Constraint 1: Each cell should be filled with a single value between 1
;;;;               and 9
;;;;
;;;; Constraint 2: Each row should contain every number from 1 to 9 once
;;;;
;;;; Constraint 3: Each column should contain every number from 1 to 9 once
;;;;
;;;; Constraint 4: Each 3x3 grid, starting from top left, should contain every
;;;;               number from 1 to 9 once

(do
  (def board-1 [[7 8 0 4 0 0 1 2 0]
                [6 0 0 0 7 5 0 0 9]
                [0 0 0 6 0 1 0 7 8]
                [0 0 7 0 4 0 2 6 0]
                [0 0 1 0 5 0 9 3 0]
                [9 0 4 0 6 0 0 0 5]
                [0 7 0 3 0 0 0 1 2]
                [1 2 0 0 0 7 4 0 0]
                [0 4 9 2 0 6 0 0 7]])

  (def board-2 [[5 3 0 0 7 0 0 0 0]
                [6 0 0 1 9 5 0 0 0]
                [0 9 8 0 0 0 0 6 0]
                [8 0 0 0 6 0 0 0 3]
                [4 0 0 8 0 3 0 0 1]
                [7 0 0 0 2 0 0 0 6]
                [0 6 0 0 0 0 2 8 0]
                [0 0 0 4 1 9 0 0 5]
                [0 0 0 0 8 0 0 7 9]])

  (def board-3 [[5 3 0 0 7 0 0 0 0]
                [6 0 0 1 9 5 0 0 0]
                [0 9 8 0 0 0 0 6 0]
                [8 0 0 0 6 0 0 0 3]
                [4 0 0 8 0 3 0 0 1]
                [7 0 0 0 2 0 0 0 6]
                [0 6 0 0 0 0 2 8 0]
                [0 0 0 4 1 9 0 0 5]
                [0 0 0 0 8 0 0 0 0]])

  (def board-4 [[0 0 0 0 0 0 0 1 2]   ;; platinum blonde
                [0 0 0 0 0 0 0 0 3]
                [0 0 2 3 0 0 4 0 0]
                [0 0 1 8 0 0 0 0 5]
                [0 6 0 0 7 0 8 0 0]
                [0 0 0 0 0 9 0 0 0]
                [0 0 8 5 0 0 0 0 0]
                [9 0 0 0 4 0 5 0 0]
                [4 7 0 0 0 6 0 0 0]])

  (defn read-board [s]
    (vector* (->> (seq s)
                  (replace {#\. #\0})
                  (map #(- (long %) (long #\0)))
```

```
                    (partition 9)
                    (map vector*))))

(defn read-boards [file]
  (->> (io/slurp-lines file)
       (map read-board)))

(defn print-board [board]
  (println)
  (->> (postwalk-replace {0 "·"} board)
       (map #(flatten (interpose "|" (partition 3 %))))
       (partition 3)
       (interpose (seq "---+---+---"))
       (flatten)
       (partition 11)
       (docoll #(apply println %))))

(defn first-empty-cell [board]
  (first (list-comp [x (range 9)
                     y (range 9)
                     :when (== 0 (get-in board [y x]))]
                    [x y])))

(defn value-not-used? [val coll]
  (nil? (some #{val} coll)))

(defn grid-3x3-vals [board x y]
  (let [xs  (-> x (/ 3) (* 3))
        ys  (-> y (/ 3) (* 3))]
    (list-comp [x1 (range xs (+ xs 3))
                y1 (range ys (+ ys 3))]
      (get-in board [y1 x1]))))

(defn possible? [board x y val]
  (and (== 0 (get-in board [y x]))                         ; cell [x y]
       (value-not-used? val (nth board y))                 ; row y
       (value-not-used? val (map #(nth % x) board))        ; col x
       (value-not-used? val (grid-3x3-vals board x y))))   ; 3x3 grid

(defn solve [board]
  (if-let [[x y] (first-empty-cell board)]
    (list-comp [v (range 1 10) :when (possible? board x y v)]
               (solve (assoc-in board [y x] v)))
    (print-board board)))


(when-not (macroexpand-on-load?)
  (print-msg-box :warn
                 """
                 macroexpand-on-load is not activated. To get a better
                 performance activate it before loading this script.

                 From the REPL run the command: !macroexpand
                 """))

(let [board board-1]
  (print-board board)
  (solve board)
  (println))
```