

```

0001 ;;;      --      --      _
0002 ;;;      \ \ / / _ _ _ _ ( _ ) _ _ _ _
0003 ;;;      \ \ / / _ _ \ ' \ | / _ _ / _ \
0004 ;;;      \ / _ _ / | | | | ( _ | _ _ /
0005 ;;;      \ / \ _ _ | _ | _ | \ _ _ \ _ _ |
0006 ;;;
0007 ;;;
0008 ;;; Copyright 2017-2022 Venice
0009 ;;;
0010 ;;; Licensed under the Apache License, Version 2.0 (the "License");
0011 ;;; you may not use this file except in compliance with the License.
0012 ;;; You may obtain a copy of the License at
0013 ;;;
0014 ;;;      http://www.apache.org/licenses/LICENSE-2.0
0015 ;;;
0016 ;;; Unless required by applicable law or agreed to in writing, software
0017 ;;; distributed under the License is distributed on an "AS IS" BASIS,
0018 ;;; WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
0019 ;;; See the License for the specific language governing permissions and
0020 ;;; limitations under the License.
0021
0022 ;;; Sudoku solver
0023
0024 ;;; Constraints for a 9x9 Sudoku
0025 ;;;
0026 ;;; Constraint 1: Each cell should be filled with a single value between 1
0027 ;;;                and 9
0028 ;;;
0029 ;;; Constraint 2: Each row should contain every number from 1 to 9 once
0030 ;;;
0031 ;;; Constraint 3: Each column should contain every number from 1 to 9 once
0032 ;;;
0033 ;;; Constraint 4: Each 3x3 grid, starting from top left, should contain every
0034 ;;;                number from 1 to 9 once
0035
0036
0037 (ns sudoku)
0038
0039 (def board-1 [[7 8 0 4 0 0 1 2 0]
0040              [6 0 0 0 7 5 0 0 9]
0041              [0 0 0 6 0 1 0 7 8]
0042              [0 0 7 0 4 0 2 6 0]
0043              [0 0 1 0 5 0 9 3 0]
0044              [9 0 4 0 6 0 0 0 5]
0045              [0 7 0 3 0 0 0 1 2]
0046              [1 2 0 0 0 7 4 0 0]
0047              [0 4 9 2 0 6 0 0 7]])
0048
0049 (def board-2 [[5 3 0 0 7 0 0 0 0]
0050              [6 0 0 1 9 5 0 0 0]
0051              [0 9 8 0 0 0 0 6 0]
0052              [8 0 0 0 6 0 0 0 3]
0053              [4 0 0 8 0 3 0 0 1]
0054              [7 0 0 0 2 0 0 0 6]
0055              [0 6 0 0 0 0 2 8 0]
0056              [0 0 0 4 1 9 0 0 5]
0057              [0 0 0 0 8 0 0 7 9]])
0058
0059 (def board-3 [[5 3 0 0 7 0 0 0 0]
0060              [6 0 0 1 9 5 0 0 0]
0061              [0 9 8 0 0 0 0 6 0]
0062              [8 0 0 0 6 0 0 0 3]
0063              [4 0 0 8 0 3 0 0 1]
0064              [7 0 0 0 2 0 0 0 6]
0065              [0 6 0 0 0 0 2 8 0]
0066              [0 0 0 4 1 9 0 0 5]
0067              [0 0 0 0 8 0 0 0 0]])
0068
0069 (def board-4 [[0 0 0 0 0 0 0 1 2] ;; platinum blonde
0070              [0 0 0 0 0 0 0 0 3]
0071              [0 0 2 3 0 0 4 0 0]
0072              [0 0 1 8 0 0 0 0 5]
0073              [0 6 0 0 7 0 8 0 0]
0074              [0 0 0 0 0 9 0 0 0]
0075              [0 0 8 5 0 0 0 0 0]
0076              [9 0 0 0 4 0 5 0 0]
0077              [4 7 0 0 0 6 0 0 0]])
0078
0079 (defn read-board [s]
0080   (vector* (-> (seq s)

```

```

0081         (replace {#\. #\0})
0082         (map #(- (long %) (long #\0)))
0083         (partition 9)
0084         (map vector*))))
0085
0086 (defn read-boards [file]
0087   (->> (io/slurp-lines file)
0088     (map read-board)))
0089
0090 (defn print-board [board]
0091   (println)
0092   (->> (postwalk-replace {0 "."} board)
0093     (map #((flatten (interpose "|" (partition 3 %))))
0094       (partition 3)
0095       (interpose (seq "----+---+---")
0096         (flatten)
0097         (partition 11)
0098         (docoll #(apply println %))))))
0099
0100 (defn first-empty-cell [board]
0101   (first (list-comp [x (range 9)
0102     y (range 9)
0103     :when (== 0 (get-in board [y x]))]
0104     [x y])))
0105
0106 (defn value-not-used? [val coll]
0107   (nil? (some #{val} coll)))
0108
0109 (defn grid-3x3-vals [board x y]
0110   (let [xs (-> x (/ 3) (* 3))
0111     ys (-> y (/ 3) (* 3))]
0112     (list-comp [x1 (range xs (+ xs 3))
0113       y1 (range ys (+ ys 3))]
0114       (get-in board [y1 x1])))
0115
0116 (defn possible? [board x y val]
0117   (and (== 0 (get-in board [y x])) ; cell [x y]
0118     (value-not-used? val (nth board y)) ; row y
0119     (value-not-used? val (map #(nth % x) board)) ; col x
0120     (value-not-used? val (grid-3x3-vals board x y)))) ; 3x3 grid
0121
0122 (defn solve [board]
0123   (if-let [[x y] (first-empty-cell board)]
0124     (list-comp [v (range 1 10) :when (possible? board x y v)]
0125       (solve (assoc-in board [y x] v)))
0126     (print-board board)))
0127
0128
0129 (when-not (macroexpand-on-load?)
0130   (println *err*
0131     ""
0132     "-----
0133     Warning: macroexpand-on-load is not activated. To get a better
0134     performance activate it before loading this script.
0135     From the REPL run the command: !macroexpand
0136     -----
0137     "")))
0138
0139 (let [board board-1]
0140   (print-board board)
0141   (solve board)
0142   (println))
0143

```