

```

0001 ;;;      --      --      -
0002 ;;;      \ \ / / _ _ _ _ ( _ ) _ _ _ _
0003 ;;;      \ \ / / _ _ ' _ \ | / _ _ / _ \
0004 ;;;      \ / _ _ / | | | | ( _ | _ _ /
0005 ;;;      \ / \ _ _ | _ | _ | \ _ _ \ _ _ |
0006 ;;;
0007 ;;;
0008 ;;; Copyright 2017-2024 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 (do
0037   (def board-1 [[7 8 0 4 0 0 1 2 0]
0038                 [6 0 0 0 7 5 0 0 9]
0039                 [0 0 0 6 0 1 0 7 8]
0040                 [0 0 7 0 4 0 2 6 0]
0041                 [0 0 1 0 5 0 9 3 0]
0042                 [9 0 4 0 6 0 0 0 5]
0043                 [0 7 0 3 0 0 0 1 2]
0044                 [1 2 0 0 0 7 4 0 0]
0045                 [0 4 9 2 0 6 0 0 7]])
0046
0047   (def board-2 [[5 3 0 0 7 0 0 0 0]
0048                 [6 0 0 1 9 5 0 0 0]
0049                 [0 9 8 0 0 0 0 6 0]
0050                 [8 0 0 0 6 0 0 0 3]
0051                 [4 0 0 8 0 3 0 0 1]
0052                 [7 0 0 0 2 0 0 0 6]
0053                 [0 6 0 0 0 0 2 8 0]
0054                 [0 0 0 4 1 9 0 0 5]
0055                 [0 0 0 0 8 0 0 7 9]])
0056
0057   (def board-3 [[5 3 0 0 7 0 0 0 0]
0058                 [6 0 0 1 9 5 0 0 0]
0059                 [0 9 8 0 0 0 0 6 0]
0060                 [8 0 0 0 6 0 0 0 3]
0061                 [4 0 0 8 0 3 0 0 1]
0062                 [7 0 0 0 2 0 0 0 6]
0063                 [0 6 0 0 0 0 2 8 0]
0064                 [0 0 0 4 1 9 0 0 5]
0065                 [0 0 0 0 8 0 0 0 0]])
0066
0067   (def board-4 [[0 0 0 0 0 0 0 1 2] ;; platinum blonde
0068                 [0 0 0 0 0 0 0 0 3]
0069                 [0 0 2 3 0 0 4 0 0]
0070                 [0 0 1 8 0 0 0 0 5]
0071                 [0 6 0 0 7 0 8 0 0]
0072                 [0 0 0 0 0 9 0 0 0]
0073                 [0 0 8 5 0 0 0 0 0]
0074                 [9 0 0 0 4 0 5 0 0]
0075                 [4 7 0 0 0 6 0 0 0]])
0076
0077   (defn read-board [s]
0078     (vector* (->> (seq s)
0079                   (replace {#\. #\0})
0080                   (map #(- (long %) (long #\0)))))

```

```

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

```