```
(require racket/trace)
(require racket/string)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;   Part III -  Battleship!

;; A simple version of the classic game, and a great exercise in list manipulation.

;; Define the only data structures used by the whole program
; the invisible ocean containing the ships of computer player
(define cOcean '((- - - - - - - - - -)
                 (- - - - - - - - - -)
                 (- - - - - - - - - -)
                 (- - - - - - - - - -)
                 (- - - - - - - - - -)
                 (- - - - - - - - - -)))


; the ocean containing the players ships.
(define pOcean '((- - - - - - - - - -)
                 (- - - - - - - - - -)
                 (- - - - - - - - - -)
                 (- - - - - - - - - -)
                 (- - - - - - - - - -)
                 (- - - - - - - - - -)))



 ;Defines ships placed for each player.
;; Format: ( (A 2) (B 3) etc), ie specifying symbol and hit value of each ship.
;; A-D are supported for ship symbols, so max of four ships. Cant have two ships of same symbol.
(define pShiplist '((A 4) (B 3) (D 2)))
(define cShiplist '((B 3) (D 2) (A 4)))

;;;;;;;;;;;;;;;;;;;;
;; THE ABOVE ARE THE ONLY GLOBAL VARIABLES ALLOWED!  THIS IS "THE STATE" OF THE GAME.
;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;
;; Functions for randomly placing the given ships. Tough stuff, but recursion makes it simpler
;;;;

;; Placeships is a front end function for randomly placing a list of ships.
;; shipslist must valid shiplist.  Called once to set up boards at start.
(define placeships
  (lambda (shiplist Ocean)
    (cond ((and (list? shiplist) (test-ships shiplist))
           (do-placements (shuffle shiplist) (shuffle (boardlist 53)) Ocean))
          (else (error "Bad Shiplists. No action taken")))))

;; do-placements is highly recursive. It randomly looks for ship placements for all ships.
;; Helper for Placeships.  Returns new board if it's possible to place all ships, #f if not
;; Uses dynamic programming to try out (if needed) every possible placement in random order.
(define do-placements
  (lambda (shiplist positions aboard)
    (cond ((null? shiplist) aboard) ; no more ships to place. We're done! Return new board.
          ((null? positions) #f) ; if we're out of positions, we failed to place all ships.
Fail.
          (else (let* ((tryit (place-one (car shiplist) (car positions) aboard))) ; try to
place first ship at first random pos.
                  (if tryit (do-placements (cdr shiplist) (cdr positions) tryit)
                      (do-placements shiplist (cdr positions) aboard)))))))
```

```
;; Place-one is a helper for do-placements and just tries to place one ship at the given board
position.
;; Figures out the random row,col to try, then passes to putship to actually try to
;; put the ship there. If putship succeeds, it returns the new board. Else #f
(define place-one
  (lambda (ship position aboard)
    (let* ((r (quotient position 9))
           (col (+ (- position (* r 9)) 1))
           (row (+ r 1))
           (shipsym (car ship))
           (shipsize (cadr ship)))
      (putship shipsym shipsize row col aboard))))

;;put-ship is the helper for place-one. Tries to place the ship at given row and col
;; returns the new board if it was possible, #f otherwise.
(define putship
  (lambda (sym size row col aboard)
    (if (= size 0) aboard ;whole ship placed. done.
        (let* ((whatsthere (getpos row col aboard)))
          (if (and whatsthere (eq? whatsthere '-))
              (putship sym (- size 1) row (+ col 1) (markchar row col sym aboard))
              #f)))))


;; Just a utility function to check is given shiplist (in pShiplist or cShiplis) is valid.
;; Initially written to support people entering their own shiplists interactively.
;; tests that it only contains integers between 2 and 4...the valid hit sizes for ships.
;; And that ship symbols are from A-D.  returns #t if good, #f if not
(define test-ships
  (lambda (shiplist)
    (cond ((null? shiplist) #t) ;done, no more ships to test
          ((not (list? (car shiplist))) #f)
          (else (let* ((tryship (car shiplist))
                       (shipsym (car tryship))
                       (shipval (cadr tryship))); grab first ship in list to test
                  (if (or (not (number? shipval))
                          (> shipval 4) (< shipval 2)
                          (not (memq shipsym '(A B C D)))) #f
                                                           (test-ships (cdr shiplist)))))))))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; The WARFARE functions!  Basically there is one action in this game: drop a bomb.
;; The key function for this is bomb, plus a bunch of helpers to see if boats are sunk
;; remove dead boats (by writing as 'sss'), and checking for a win.
;;;;;;;;

;; The main action function in the game:  it drops a bomb at a given location.
;; Takes location (r,c) plus 'P or 'C designating whether computer or player board is being
bombed.
;; It figures out what's under that location, marks the hit or miss (updates board), reports it
to the display,
;; checks for sunk ships (which involves another update to mark the ship sunk), and checks for
win/loss.
;; Game play = calling this function repeatedly, with P and C alternating as turns are taken.
(define bomb
  (lambda (r c bombee)

;; *** you need to write this, plus any helper functions you want ****
;; *** Obviously your bomb (or related helpers) will want to set! the players
;; board, and maybe (if ships sunk) that players shipslist.  That way winning
```

```
;; can be noticed simply:  you lose if your shipslist ends up null.
     )))

;;; *** I wrote about 4-5 helper functions to go with "bomb" to take care of various things
;; that need to happen in bomb, i.e., figure out whats under the bomb, marking the board,
;; and checking for wins.

;; Here is the header for one of those functions, called markchar.  I'm giving you the header
;; for it because it is the one "warfare" function also called elsewhere in my code (i.e. while
;; placing the ship characters on the original boards.  So, just as with bomb, you will need to
;; implement exactly this function header so the code as a whole will run.
;; In fact, I would recommend you start by writing markchar; it's a simple function that'll
help
;; you get the feel for what you're doing.

; Markchar is a generally useful function used to place a character on a board. Takes in a row,
col,
;; and an Ocean (board) to work on, plus the symbol you want to place at that location.  It
first checks
;; to see if row,col is valid (returns #f if not), and then recursively digs down into the
board to place
;; the given character at the target location.  It returns the Ocean with the given character
updated.
;; Used when placing ships, dropping bombs, etc, whenever you need to place a character on the
board.
(define markchar
  (lambda (r c char Ocean)




;;;;;;;;;;;;;;;;;
;; INTERACTIVITY.  Just one play function to make life fluid.  All it does is gets input from
user,
;; bombs that location, fires its own random shot, and shows the resulting state. Then repeats
by recursing...

;; Call this to play fluidly.  Handles turn taking, gets user input,
;; and displays boards between turns.
(define play
  (lambda ()
    (let* ((prompt (displayln "Enter your shot as: row col"))
           (in (read-line))
           (rc (get-ints in))
           (shot (if rc (bomb (car rc) (cadr rc) 'C)))
           (randshot (make-rc (random 54)))
           (engarde (ndisplay "My turn! I'm bombing "
                              (list (car randshot) (cadr randshot)) " "))
           (counter (if rc (bomb (car randshot) (cadr randshot) 'P)))
           (stat (stat)))
      (if rc (play) (displayln "Thanks for playing")))))


;;;;;;;;;;;;;;;;;;;;;;;;
;;; SOME BORING HELPER FUNCTIONS. Bunch of housekeeping help.

;; Basically the viewing function. It just prints some initial header stuff
;; then calls the recursive function showOceans which traverses the two
;; boards in parallel printing rows as a side effect.
(define stat
  (lambda ()
    (let ((first (newline))
```

```
          (foo (ndisplay "   Opponent"  "         "  "Your ships"))
          (fie (ndisplay "  ----------" "        " "-----------"))
          (foe (ndisplay '(\ 123456789) "      " '(\ 123456789))))
      (showOceans cOcean pOcean 1 #t))))

;;cheater fn for testing, shows ships on computers board
(define cheat
  (lambda ()
    (let ((foo (displayln "CHEATER! Ok heres my secret board:")))
      (showOceans cOcean  pOcean 1 #f))))

;; just creates a list of ints from pos to zero, numbered from high to low. Used to
;; create a list of possible board positions for placeships.
(define boardlist
  (lambda (pos)
    (cond ((= pos 0) '(0))
          (else (cons pos (boardlist (- pos 1)))))))

;; Horrifically inefficient but simple and compact list shuffler.
;; Takes in a list, returns the randomly shuffled list
(define shuffle
  (lambda (list)
    (if (< (length list) 2)
        list
        (let ((item (list-ref list (random (length list)))))
          (cons item (shuffle (remove item list)))))))

;; Just recursively digs down and gets the contents of (row,col)  on given board.
;;; returns #f if bad r,c; else returns the symbol at target position.
(define getpos
  (lambda (r c Ocean)
    (cond ((or (= r 0) (null? Ocean)) #f)
          ((= r 1) (getcolpos c (car Ocean)))
          (else (getpos (- r 1) c (cdr Ocean))))))

;; helper for getpos. Once I have a row, getcolpos recursively scans downs the cols and
;; returns the requested position.
(define getcolpos
  (lambda (c row)
    (cond ((or (= c 0) (null? row)) #f)
          ((= c 1) (car row))
          (else (getcolpos (- c 1) (cdr row))))))

; A function used to show status. Compresses row lists strings for compactness.
;;The trick here is that you need to show the row/col labels of the board too.
;; if hide? is true, it means I'm displaying computer board for player
;; and need to hide ships position and type
;; Takes the two boards (computer and player, a starting rowlabel, and hide?
;; hide? is t/f.  If true, it will hide key data in computer's board.  Else
;; it shows it like it is.
(define showOceans
  (lambda (cboard pboard row hide?)
    (if (null? cboard) (ndisplay '(\ 123456789) "      " '(\ 123456789) )
        (let* ((rowlabel (letnum row))
               (newrow (if hide? (cleanrow (car cboard)) (car cboard)))
               (foo (ndisplay rowlabel (list (slist->string newrow)) "      " rowlabel (list
(slist->string (car pboard))) )))
          (showOceans (cdr cboard) (cdr pboard) (+ row 1) hide?)))))

;; Cleanrow prepares a row for display. If you're displaying a row of computer board
;; for the player, the obviously you need to hide (ie delete) all ships, and anonymize
;; the hits (turn them into X's) so you can't tell what kind of ship it is.
;; This guy just takes in a given row, cleans it up, and returns it for display.
```

```
(define cleanrow
  (lambda (arow)
    (cond ((null? arow) arow)
          ((memq (car arow) '(a b c d)) (cons 'X (cleanrow (cdr arow)))) ; anonymize hits
          ((memq (car arow) '(A B C D)) (cons '- (cleanrow (cdr arow)))) ; hide non-hit ships
          (else (cons (car arow) (cleanrow (cdr arow)))))))

;Checks if r and c are valid rows and cols on a board. r can be either a-f or 1-6
;; returns #t if valid position, #f otherwise
(define checkvalid
  (lambda (r c)
    (let ((row (if (symbol? r) (letnum r) r)))
      (and row (number? c) (> row 0) (< row 7) (> c 0) (< c 10)))))

;;make-rc takes in an int position value (0-53) on the board,
;; transforms and returns as corresponding list of (row col)
(define make-rc
  (lambda (pos)
    (let* ((r (quotient pos 9))
          (col (+ (- pos (* r 9)) 1))
          (row (+ r 1)))
      (list row col))))


;; helper that turns a letter into a number or vice versa.
;; Used to convert "a b c" labels to ints; or 1-6 row labels to ints.
;; Allows users flexibility on how they input coords.
;;returns #f is out of range
(define letnum
  (lambda (val)
    (let ((result (if (number? val)
                      (assoc val '((1 a) (2 b) (3 c) (4 d) (5 e) (6 f)))
                      (assoc val '((a 1) (b 2) (c 3) (d 4) (e 5) (f 6))))))
      (if result (cadr result) #f))))

;; Just takes a list of symbols and mashes them together into a nice compact string
;; Note that it uses string-join, which is imported from (racket/string) at top
(define slist->string
  (lambda (alist)
    (string-join (map symbol->string alist) "")))

;; Hit? is just super simple:  returns true if input is in (a b c d x X) and false if O
(define hit?
  (lambda (hitchar)
    (if (memq hitchar '(a b c d x X)) #t #f)))

;; Simple list-replce. just replace OUT with IN throughout alist and return result
(define replacechar
  (lambda (out in alist)
    (cond ((null? alist) alist)
          ((eq? (car alist) out) (cons in (replacechar out in (cdr alist))))
          (else (cons (car alist) (replacechar out in (cdr alist)))))))

;; Displays multiple items followed by newline. Avoids clutter.
(define ndisplay
  (lambda alist
    (nd-h alist)))
; Helper for ndisplay
(define nd-h (lambda (alist)
    (if (null? alist) (displayln " ")
        (begin (display (car alist)) (nd-h (cdr alist))))))

;; Converts string input from read-line to a (row col) list
```

```
;; Used when getting user input in (play) fn.  What a pain.
(define get-ints
  (lambda (str)
    (if (or (string-ci=? str "q") (string-ci=? str "")) #f
        (let* ((alist (string->list str))
               (ints (int-list alist)))
          (list (first ints) (last ints))))))

;;Helper for get-ints. Goes through a charlist and turns into ints.
;; Used to process user input read in.
(define int-list
  (lambda (charlist)
    (if (null? charlist) '()
    (let* ((charnum (char->integer (car charlist)))
           (charval (if (> charnum 60) (- charnum 96) (- charnum 48))))
          (cons charval (int-list (cdr charlist)))))))

;; Just returns the last item in a list.  Usually built it, but not in Racket
(define last (lambda (alist)
     (cond ((null? alist) alist)
           ((null? (cdr alist)) (car alist))
           (else (last (cdr alist))))))



;; Set up the game board to play!
(set! pOcean (placeships pShiplist pOcean))
(set! cOcean (placeships cShiplist cOcean))
(stat)
```