```
;; CS 135 :: Fall 2017 :: Posted solution :: A09 :: ca.rkt


;; A Bit is (anyof 0 1)


;; ------ Q1a ------

;; (apply-rule a b c rule) applies a standardized cellular automata
;;   rule to the consecutive cells a, b, and c
;; apply-rule: Bit Bit Bit Nat -> Bit
;; requires: rule <= 255
;; Examples:
(check-expect (apply-rule 1 0 1 110) 1)
(check-expect (apply-rule 0 1 1 102) 0)

(define (apply-rule a b c rule)
  (remainder (floor (/ rule (expt 2 (+ (* 4 a) (* 2 b) c)))) 2))

;; Tests:
(check-expect (apply-rule 1 1 1 30) 0)
(check-expect (apply-rule 0 0 1 30) 1)
(check-expect (apply-rule 1 0 0 60) 1)
(check-expect (apply-rule 1 0 0 158) 1)


;; ------ Q1b ------

;; (next-row row rule) applies a standardized cellular automata
;;   rule to a row of consecutive cells
;; next-row: (listof Bit) Nat -> (listof Bit)
;; requires: rule <= 255
;;           row is non-empty
;; Examples:
(check-expect (next-row '(0) 30) '(0))
(check-expect (next-row '(0 1 0 1) 60) '(0 1 1 1))

(define (next-row row rule)
  (local
    [;; (transform-cells row) applies a standardized cellular automata
     ;;   rule to a row of consecutive cells
     ;; transform-cells: (listof Bit) -> (listof Bit)
     (define (transform-cells row)
       (cond
         [(empty? (rest (rest row))) empty]
         [else (cons (apply-rule (first row) (second row) (third row) rule)
                     (transform-cells (rest row)))]))]
    (transform-cells (append '(0) row '(0)))))

;; Tests:
(check-expect (next-row '(0 0 0 0 0) 1) '(1 1 1 1 1))
(check-expect (next-row '(0 0 1 0 0) 30) '(0 1 1 1 0))
(check-expect (next-row '(0 0 1 0 1 0 0) 94) '(0 1 1 0 1 1 0))
(check-expect (next-row '(1 1 0 1 1) 190) '(1 0 1 1 0))


;; ------ Q1c ------
```

```
;; (iterate f base n) produces a sequence of n values:
;;    base, (f base), (f (f base)), ..., (f ... (f base))), where the kth
;;    element consists of f applied k-1 times to base.
;; iterate: (X -> X) X Nat -> (listof X)
;; Examples:
(check-expect (iterate identity 1 0) '())
(check-expect (iterate sqr 2 5) '(2 4 16 256 65536))

(define (iterate f base n)
  (cond
    [(zero? n) empty]
    [else (cons base (iterate f (f base) (sub1 n)))]))

;; Tests:
(check-expect (iterate add1 0 8) '(0 1 2 3 4 5 6 7))
(check-expect (iterate (lambda (x) (cons 0 x)) empty 4)
              '(() (0) (0 0) (0 0 0)))


;; ------ Q1d ------

;; (run-automaton row rule n) produces the result of successively applying
;;    a standardized cellular automata rule to a given row n-1 times
;; run-automaton: (listof Bit) Nat Nat -> (listof (listof Bit))
;; requires: rule <= 255
;; Examples:
(check-expect (run-automaton '(0 1 0 1 0) 99 1) '((0 1 0 1 0)))
(check-expect (run-automaton '(0 1 0) 30 4)
              '((0 1 0) (1 1 1) (1 0 0) (1 1 0)))

(define (run-automaton row rule n)
  (iterate (lambda (row) (next-row row rule)) row n))

;; Tests:
(check-expect (run-automaton '(0 0 0 1 0 0 0) 54 5)
              '((0 0 0 1 0 0 0)
                (0 0 1 1 1 0 0)
                (0 1 0 0 0 1 0)
                (1 1 1 0 1 1 1)
                (0 0 0 1 0 0 0)))
(check-expect (run-automaton '(0 0 1 1 0 0 0) 60 6)
              '((0 0 1 1 0 0 0)
                (0 0 1 0 1 0 0)
                (0 0 1 1 1 1 0)
                (0 0 1 0 0 0 1)
                (0 0 1 1 0 0 1)
                (0 0 1 0 1 0 1)))



;; CS 135 :: Fall 2017 :: Posted solution :: A09 :: rectangle.rkt


(require "rectanglelib.rkt")

(define-struct cell (num used?))
;; A Cell is a (make-cell Nat Bool)
```

```racket
;; A Grid is a (listof (listof Cell))
;; requires: the grid contains a non-empty list of non-empty lists,
;;           all the same length.

(define-struct rect (x y w h))
;; A Rect is a (make-rect Nat Nat Nat Nat)

(define-struct state (grid rects))
;; A State is a (make-state Grid (listof Rect))

(define puzz '((0 0 0 0 0 5 0)
               (0 0 0 0 0 2 2)
               (0 3 0 6 3 2 0)
               (4 0 0 0 0 0 0)
               (0 0 0 4 0 4 0)
               (2 0 6 0 2 4 0)
               (0 0 0 0 0 0 0)))

(define big-puzz '((4 0 7 0 0 0 0 0 0 0 0 21 0)
                   (0 3 2 0 0 0 0 0 0 0 0 0 2)
                   (0 0 0 0 0 0 0 2 3 0 0 0 0)
                   (0 0 0 20 0 0 0 0 0 0 0 0 5)
                   (0 2 0 0 0 0 0 4 0 0 0 0 0)
                   (0 0 3 0 0 0 0 0 0 0 0 0 0)
                   (3 0 0 0 0 5 2 4 0 0 0 0 0)
                   (0 0 0 0 0 2 0 6 0 0 0 0 0)
                   (0 0 0 20 0 0 0 0 0 0 0 0 0)
                   (0 0 0 0 0 0 0 0 0 0 0 0 0)
                   (0 0 0 0 0 0 0 0 0 0 0 24 0)
                   (0 0 0 0 4 0 4 0 0 0 4 0 0)
                   (0 0 3 0 0 0 0 0 0 0 8 0 2)))

;; Some cell constants for testing
(define c0 (make-cell 0 false))
(define c1 (make-cell 1 false))
(define c2 (make-cell 2 false))
(define c3 (make-cell 3 false))
(define c4 (make-cell 4 false))
(define c0t (make-cell 0 true))
(define c1t (make-cell 1 true))
(define c2t (make-cell 2 true))
(define c3t (make-cell 3 true))
(define c4t (make-cell 4 true))


;; ------ Q2a ------

;; (map2d f lolst) applies f to every element in lolst
;; map2d: (X -> Y) (listof (listof X)) -> (listof (listof Y))
;; Examples:
(check-expect (map2d identity empty) empty)
(check-expect (map2d sqr '((1 2 3) (4 5 6))) '((1 4 9) (16 25 36)))

(define (map2d f lolst)
  (map (lambda (row) (map f row)) lolst))

;; Tests:
(check-expect (map2d identity '(() () ())) '(() () ()))
```

```
(check-expect (map2d symbol? '((0 1 a) (b c 0) (99 d 99)))
              (list (list false false true)
                    (list true true false)
                    (list false true false)))


;; ------ Q2b ------

;; (construct-puzzle lolst) produces a state with a grid corresponding
;;    to the elements of lolst and no filled-in rectangles
;; construct-puzzle: (listof (listof Nat)) -> State
;; requires: lolst is a non-empty list of non-empty lists, all the same length.
;; Example:
(check-expect (construct-puzzle '((1)))
              (make-state (list (list c1)) empty))

(define (construct-puzzle lolst)
  (make-state (map2d (lambda (x) (make-cell x false)) lolst) empty))

;; Tests:
(check-expect (construct-puzzle '((2 0) (0 2)))
              (make-state (list (list c2 c0) (list c0 c2)) empty))
(check-expect (construct-puzzle '((0 1 2) (0 0 3) (3 0 0)))
              (make-state (list (list c0 c1 c2)
                                (list c0 c0 c3)
                                (list c3 c0 c0)) empty))

;; ------ Q2c ------

;; (solved? st) determines if st represents a solved puzzle
;; solved?: State -> Bool
;; Examples:
(check-expect (solved? (construct-puzzle '((1)))) false)
(check-expect (solved? (make-state
                        (list (list c2t c1t) (list c0t c1t))
                        (list (make-rect 0 0 1 2)
                              (make-rect 0 1 1 1)
                              (make-rect 1 1 1 1)))) true)

(define (solved? st)
  (local
    [(define used-grid (map2d cell-used? (state-grid st)))
     (define flat-used-grid (foldr append empty used-grid))]
    (not (member? false flat-used-grid))))

;; Note that the list of rectangles in the State doesn't affect the answer,
;; so we'll shorten these tests by keeping them empty.
;; Tests:
(check-expect
 (solved? (make-state
           (list (list c0t c0t c0t c0t)
                 (list c0t c0t c0  c0t)
                 (list c0t c0t c0t c0t)
                 (list c0t c0t c0t c0t)) empty)) false)
(check-expect
 (solved? (make-state
           (list (list c0t c1t c2t c3t)
                 (list c1t c2t c3t c0t)
```

```
              (list c2t c3t c0t c1t)
              (list c3t c0t c1t c2t)) empty)) true)


;; ------ Q2d ------

;; (get-first-unused grid) finds the topmost, leftmost coordinate in the grid
;;   of a cell that isn't marked as used
;; get-first-unused: Grid -> (list Nat Nat)
;; requires: at least one unused Cell exists in grid
;; Examples:
(check-expect (get-first-unused (list (list c0)))
              (list 0 0))
(check-expect (get-first-unused
               (list (list c0t c0t c0t c0t)
                     (list c0t c0t c0t c0t)
                     (list c0t c0  c0t c0t)))
              (list 1 2))

(define (get-first-unused grid)
  (local
    [;; (first-unused/row cnum rnum loc) finds the position of the first unused
     ;;   cell in the rnum'th row, loc, after already seeing cnum elements
     ;; first-unused/row: Nat Nat (listof Cell) -> (anyof false (list Nat Nat))
     (define (first-unused/row cnum rnum loc)
       (cond
         [(empty? loc) false]
         [(not (cell-used? (first loc))) (list cnum rnum)]
         [else (first-unused/row (add1 cnum) rnum (rest loc))]))

     ;; (search-rows rnum lolst) finds the position of the first unused
     ;;   cell lolst after already seeing rnum rows
     ;; search-rows: Nat (listof (listof Cell)) -> (anyof false (list Nat Nat))
     ;; requires: at least one unused Cell exists in lolst
     (define (search-rows rnum lolst)
       (local
         ;; No need to check for an empty list -- we assume we'll find an unused Cell
         [(define first-row (first-unused/row 0 rnum (first lolst)))]
         (cond
           [(boolean? first-row) (search-rows (add1 rnum) (rest lolst))]
           [else first-row])))]
    (search-rows 0 grid)))

;; Tests:
(check-expect (get-first-unused
               (list (list c0t c0)
                     (list c0 c0)))
              (list 1 0))
(check-expect (get-first-unused
               (list (list c0t c0t)
                     (list c0 c0)))
              (list 0 1))
(check-expect (get-first-unused
               (list (list c0t c0t)
                     (list c0t c0)))
              (list 1 1))
```

```
;; ------ Q2e ------

;; (subgrid->list lolst bounds) produces a list of the elements in lolst
;;   that lie within a given bounds
;; subgrid->list: (listof (listof X)) Rect -> (listof X)
;; requires: The rectangle is completely contained within the grid
;; Examples:
(check-expect (subgrid->list '((1)) (make-rect 0 0 1 1)) '(1))
(check-expect (subgrid->list '((1 2 3 4) (5 6 7 8) (9 10 11 12))
                             (make-rect 1 1 2 2))
              '(6 7 10 11))

(define (subgrid->list lolst bounds)
  (local
    [;; (sublist lst start w) produces the first w elements in lst
     ;;   after a start position
     ;; sublist: (listof X) Nat Nat -> (listof X)
     ;; requires: start + w <= (length lst)
     (define (sublist lst start w)
       (cond
         [(> start 0) (sublist (rest lst) (sub1 start) w)]
         [(> w 0) (cons (first lst) (sublist (rest lst) 0 (sub1 w)))]
         [else empty]))]
    (foldr append empty
           (map (lambda (row) (sublist row (rect-x bounds) (rect-w bounds)))
                (sublist lolst (rect-y bounds) (rect-h bounds))))))

;; Tests:
(check-expect (subgrid->list '() (make-rect 0 0 0 0)) '())
(check-expect (subgrid->list '((1 2 3 4) (5 6 7 8) (9 10 11 12))
                             (make-rect 0 1 3 2))
              '(5 6 7 9 10 11))


;; (map-subgrid f lolst bounds) applies f to each element of lolst
;;   that lie within a given bounds
;; map-subgrid: (X -> Y) (listof (listof X)) -> (listof (listof (anyof X Y)))
;; requires: The rectangle is completely contained within the grid
;; Examples:
(check-expect (map-subgrid sqr '((2)) (make-rect 0 0 1 1)) '((4)))
(check-expect (map-subgrid number->string
                           '((1 2 3) (4 5 6) (7 8 9))
                           (make-rect 2 1 1 2))
              '((1 2 3) (4 5 "6") (7 8 "9")))

(define (map-subgrid f lolst bounds)
  (local
    [;; (map-sublist f lst start w) applies f to the first w elements of lst
     ;;   after a start position
     ;; map-sublist: (X -> Y) (listof X) Nat Nat -> (listof (anyof X Y))
     ;; requires: start + w <= (length lst)
     (define (map-sublist f lst start w)
       (cond
         [(> start 0) (cons (first lst) (map-sublist f (rest lst) (sub1 start) w))]
         [(> w 0) (cons (f (first lst)) (map-sublist f (rest lst) 0 (sub1 w)))]
         [else lst]))]
    (map-sublist
     (lambda (row) (map-sublist f row (rect-x bounds) (rect-w bounds)))
```

```
     lolst (rect-y bounds) (rect-h bounds))))

;; Tests:
(check-expect (map-subgrid identity '() (make-rect 0 0 0 0)) '())
(check-expect (map-subgrid sqr '((1 2 3) (4 5 6) (7 8 9))
                           (make-rect 0 0 2 3))
              '((1 4 3) (16 25 6) (49 64 9)))


;; (neighbours st) produces all states that can be reached by adding
;;   a single rectangle to st
;; neighbours: State -> (listof State)
;; Examples:
(check-expect (neighbours (make-state (list (list c2)) '())) empty)
(check-expect (neighbours (make-state (list (list c2 c0) (list c0 c1)) '()))
              (list (make-state (list (list c2t c0t) (list c0 c1))
                                (list (make-rect 0 0 2 1)))
                    (make-state (list (list c2t c0) (list c0t c1))
                                (list (make-rect 0 0 1 2)))))

(define (neighbours st)
  (local
    [(define grid (state-grid st))
     (define unused (get-first-unused grid))
     (define width (length (first grid)))
     (define height (length grid))

     (define max-rwidth (- width (first unused)))
     (define max-rheight (- height (second unused)))

     (define all-rects
       (foldr append empty
              (build-list max-rheight
                          (lambda (h)
                            (build-list max-rwidth
                                        (lambda (w)
                                          (make-rect (first unused)
                                                     (second unused)
                                                     (add1 w)
                                                     (add1 h))))))))

     ;; (valid-rect? rec) determines if rec can legally be added to st
     ;; valid-rect?: Rect -> Bool
     (define (valid-rect? rec)
       (local
         [(define cells (subgrid->list grid rec))
          (define pos (filter positive? (map cell-num cells)))]
         (and
          (not (member? true (map cell-used? cells)))
          (= (length pos) 1)
          (= (first pos) (* (rect-w rec) (rect-h rec))))))]

    (map (lambda (rec)
           (make-state
            (map-subgrid (lambda (c) (make-cell (cell-num c) true))
                         grid rec)
            (cons rec (state-rects st))))
         (filter valid-rect? all-rects))))
```

```
;; Tests:
(check-expect (neighbours (make-state (list (list c0 c2) (list c2 c0)) '()))
              (list (make-state (list (list c0t c2t) (list c2 c0))
                                (list (make-rect 0 0 2 1)))
                    (make-state (list (list c0t c2) (list c2t c0))
                                (list (make-rect 0 0 1 2)))))

(check-expect (neighbours (make-state (list (list c4 c0) (list c0 c2)) '())) '())

(check-expect (neighbours (make-state (list (list c0t c2t c0t)
                                            (list c0 c0 c2t)
                                            (list c0 c4 c1))
                                      (list (make-rect 0 0 2 1)
                                            (make-rect 2 0 1 2))))
              (list (make-state (list (list c0t c2t c0t)
                                      (list c0t c0t c2t)
                                      (list c0t c4t c1))
                                (list (make-rect 0 1 2 2)
                                      (make-rect 0 0 2 1)
                                      (make-rect 2 0 1 2)))))

(check-expect (neighbours (make-state (list (list c0t c2t c0t)
                                            (list c0 c0 c2t)
                                            (list c1 c4 c0))
                                      (list (make-rect 0 0 2 1)
                                            (make-rect 2 0 1 2)))) empty)


;; ------ Q2f ------

;; (solve-rectangle-puzzle puzz) produces the solution to puzz, or false
;;   if no solution exists
;; solve-rectangle-puzzle: (listof (listof Nat)) -> (anyof false (listof Rect))
;; requires: puzz is a non-empty list of non-empty lists, all the same length
;; Examples:
(check-expect (solve-rectangle-puzzle '((0 2 3) (4 0 0) (0 0 0)))
              (list (make-rect 0 1 2 2)
                    (make-rect 2 0 1 3)
                    (make-rect 0 0 2 1)))
(check-expect (solve-rectangle-puzzle '((5))) false)

(define (solve-rectangle-puzzle puzz)
  (local
    [(define soln (search solved? neighbours (construct-puzzle puzz)))]
    (cond
      [(false? soln) false]
      [else (state-rects soln)])))

;; Tests:
(check-expect (solve-rectangle-puzzle '((0 3 0) (3 3 0) (0 0 0)))
              false)

(check-expect (solve-rectangle-puzzle puzz)
              (list
               (make-rect 5 5 2 2)
               (make-rect 4 5 1 2)
               (make-rect 1 5 3 2)
```

```
                (make-rect 0 5 1 2)
                (make-rect 0 4 4 1)
                (make-rect 5 3 2 2)
                (make-rect 5 2 2 1)
                (make-rect 4 2 1 3)
                (make-rect 4 1 2 1)
                (make-rect 2 1 2 3)
                (make-rect 1 1 1 3)
                (make-rect 6 0 1 2)
                (make-rect 1 0 5 1)
                (make-rect 0 0 1 4)))

(check-expect (solve-rectangle-puzzle big-puzz)
              (list
               (make-rect 11 12 2 1)
               (make-rect 3 12 8 1)
               (make-rect 0 12 3 1)
               (make-rect 9 11 4 1)
               (make-rect 5 11 4 1)
               (make-rect 5 8 8 3)
               (make-rect 4 8 1 4)
               (make-rect 7 7 6 1)
               (make-rect 4 7 2 1)
               (make-rect 0 7 4 5)
               (make-rect 6 6 1 2)
               (make-rect 1 6 5 1)
               (make-rect 7 5 2 2)
               (make-rect 1 4 1 2)
               (make-rect 0 4 1 3)
               (make-rect 7 3 2 2)
               (make-rect 2 3 1 3)
               (make-rect 12 2 1 5)
               (make-rect 7 1 1 2)
               (make-rect 3 1 4 5)
               (make-rect 2 1 1 2)
               (make-rect 1 1 1 3)
               (make-rect 12 0 1 2)
               (make-rect 9 0 3 7)
               (make-rect 8 0 1 3)
               (make-rect 1 0 7 1)
               (make-rect 0 0 1 4)))
```