```
;; Question 2:
;; Part (a)
(define-struct line (slope intercept))
;; A Line = (make-line (union Num Symbol) Num)

;; Template:
;; my-line-fn: Line -> Any
(define (my-line-fn aline)
  (
   ... (line-slope aline) ...
   ... (line-intercept aline) ...))

;; Part (b)
;; helper function
;; compute-slope: Posn Posn -> Num
;; Purpose:
;; compute the slope between two points, assuming it is defined
;; Examples:
(check-expect (compute-slope (make-posn 0 0) (make-posn 3 3)) 1)
(check-expect (compute-slope (make-posn 3 4) (make-posn 5 7)) 3/2)
(define (compute-slope p1 p2)
  (/ (- (posn-y p1) (posn-y p2)) (- (posn-x p1) (posn-x p2))))
;; Tests:
(check-expect (compute-slope (make-posn -2 4) (make-posn 3 -7)) -11/5)
(check-expect (compute-slope (make-posn 7 6) (make-posn 9 6)) 0)

;; two-points->Line: Posn Posn -> Line
;; Purpose:  create a Line structure
;; (i.e., calculate slope and intercept)
;; based on the given points (Posn's)
;; if the line is non-vertical, the intercept is the
;; y-intercept.  If the line is vertical, the intercept
;; is the x-intercept
;; Examples:
(check-expect
 (two-points->Line (make-posn 0 0) (make-posn -2 -2))
 (make-line 1 0))
(check-expect
 (two-points->Line (make-posn 3 4) (make-posn 5 5))
 (make-line 1/2 5/2))
(define (two-points->Line p1 p2)
  (cond
    ;; check for vertical line
    [(= (posn-x p1) (posn-x p2)) (make-line 'undefined (posn-x p1))]
    ;; non-vertical
    [else (make-line
           (compute-slope p1 p2)
           (- (posn-y p1) (* (compute-slope p1 p2) (posn-x p1))))]))
;; Tests:
;; test vertical
(check-expect
 (two-points->Line (make-posn 4 3) (make-posn 4 7))
 (make-line 'undefined 4))
;; test negative slope
(check-expect
 (two-points->Line (make-posn -2 4) (make-posn 3 -7))
 (make-line -11/5 -2/5))
;; test horizontal
```

```
(check-expect
 (two-points->Line (make-posn 7 6) (make-posn 9 6))
 (make-line 0 6))

;; Part (c)
;; perp-Line: Posn Line -> Line
;; Purpose:
;; Produce a new Line which is perpendicular (i.e., at 90 degrees)
;; to the given Line, which passes through the given point
;; Examples:
(check-expect (perp-Line (make-posn 3 2) (make-line 1 0))
              (make-line -1 5))
;; Definition
(define (perp-Line point line)
  (cond
    [(equal? (line-slope line) 'undefined)
     (make-line 0 (posn-y point))]
    [(= (line-slope line) 0)
     (make-line 'undefined (posn-x point))]
    [else
     (make-line (- (/ 1 (line-slope line)))
                (- (posn-y point) (* (posn-x point) (- (/ 1 (line-slope
line))))))]))
;; Tests
(check-expect (perp-Line (make-posn 3 4) (make-line 'undefined 6))
              (make-line 0 4))
(check-expect (perp-Line (make-posn 7 8) (make-line 0 4))
              (make-line 'undefined 7))
(check-expect (perp-Line (make-posn 6 9) (make-line -11/5 -2/5))
              (make-line 5/11 69/11))
(check-expect (perp-Line (make-posn 0 0) (make-line 'undefined 0))
              (make-line 0 0))
```

```
;; Question 3
;; Part (a)
(define-struct card (suit rank))
;; A Card=(make-card Symbol Num)

;; Template:
;; my-card-fn: Card -> Any
(define (my-card-fn acard)
  (
    ... (card-rank acard) ...
    ... (card-suit acard) ...))


;; Part (b)
;; better-Card: Card Card -> Card
;; Purpose:
;; Given two cards return the card either with higher suit value
;; or, if they are the same suit, the higher card value
;; Examples:
(check-expect (better-Card (make-card 'spades 1) (make-card 'diamonds 13))
              (make-card 'spades 1))  ;; Tests Case 3
(check-expect (better-Card (make-card 'clubs 3) (make-card 'clubs 4))
              (make-card 'clubs 4)) ;; Tests Case 2
(define (better-Card c1 c2)
  (cond
    ;; if the suits are the same
    [(symbol=? (card-suit c1) (card-suit c2))
      (cond
        [(> (card-rank c1) (card-rank c2)) c1]  ;; Case 1
        [else c2])]                            ;; Case 2
    ;; if the suits are different
    [(symbol=? (card-suit c1) 'spades) c1]      ;; Case 3
    [(symbol=? (card-suit c2) 'spades) c2]      ;; Case 4
    [(symbol=? (card-suit c1) 'hearts) c1]      ;; Case 5
    [(symbol=? (card-suit c2) 'hearts) c2]      ;; Case 6
    [(symbol=? (card-suit c1) 'diamonds) c1]    ;; Case 7
    [else c2]))                                 ;; Case 8
;; Tests:
;; test Case 1
(check-expect (better-Card (make-card 'hearts 3) (make-card 'hearts 2))
              (make-card 'hearts 3))
;; test Case 3
(check-expect (better-Card (make-card 'spades 1) (make-card 'hearts 13))
              (make-card 'spades 1))
;; test Case 4
(check-expect (better-Card  (make-card 'clubs 13) (make-card 'spades 1))
              (make-card 'spades 1))
;; test Case 5
(check-expect (better-Card (make-card 'hearts 3) (make-card 'diamonds 13))
              (make-card 'hearts 3))
;; test Case 6
(check-expect (better-Card (make-card 'diamonds 13) (make-card 'hearts 3))
              (make-card 'hearts 3))
;; test Case 7
(check-expect (better-Card (make-card 'diamonds 13) (make-card 'clubs 1))
              (make-card 'diamonds 13))
;; test Case 8
(check-expect (better-Card (make-card 'clubs 1) (make-card 'diamonds 13))
```

```
              (make-card 'diamonds 13))

;; Part (c)
;; Define some cards for testing various functions
;; Define some cards (as constants) to make examples/testing easier
(define card1 (make-card 'spades 3))
(define card2 (make-card 'spades 4))
(define card3 (make-card 'spades 5))
(define card4 (make-card 'diamonds 3))
(define card5 (make-card 'clubs 3))

;; helpers:
;; same-suit?: Card Card Card -> Boolean
;; Purpose:
;; Given three cards, return true if and only if they
;; are the same suit
;; Examples:
(check-expect (same-suit? card1 card2 card3) true)
(check-expect (same-suit? card1 card3 card4) false)
(define (same-suit? c1 c2 c3)
  (and (symbol=? (card-suit c1) (card-suit c2)) (symbol=? (card-suit c2) (card-suit
c3))))
;; Tests:
;; Try all other positions of one non suited card
(check-expect (same-suit? card1 card4 card3) false)
(check-expect (same-suit? card4 card3 card1) false)
;; Try with three different suits
(check-expect (same-suit? card1 card4 card5) false)

;; all-same-num?: Num Num Num -> Boolean
;; Purpose:
;; Returns true if all three numbers are the same, false otherwise
;; Examples:
(check-expect (all-same-num? 3 3 3) true)
(check-expect (all-same-num? 3 3 4) false)
(define (all-same-num? num1 num2 num3)
  (and (= num1 num2) (= num1 num3)))
;; Tests:
(check-expect (all-same-num? 3 4 3) false)
(check-expect (all-same-num? 4 3 3) false)
(check-expect (all-same-num? 2 3 4) false)

;; consecutive-num?: Num Num Num -> Boolean
;; Purpose:
;; Returns true if all three numbers are consecutive, false otherwise
;; Examples:
(check-expect (consecutive-num? 6 7 8) true)
(check-expect (consecutive-num? 2 4 6) false)
;; Definition
(define (consecutive-num? num1 num2 num3)
  (and (= (- (max num1 num2 num3) (min num1 num2 num3)) 2)
       (= (- (max num1 num2 num3) 1) (/ (+ num1 num2 num3) 3))))
;; Tests:
(check-expect (consecutive-num? 1 2 3) true)
(check-expect (consecutive-num? 1 3 2) true)
(check-expect (consecutive-num? 2 1 3) true)
(check-expect (consecutive-num? 2 3 1) true)
(check-expect (consecutive-num? 3 2 1) true)
```

```
(check-expect (consecutive-num? 3 1 2) true)
(check-expect (consecutive-num? 3 4 6) false)
(check-expect (consecutive-num? 3 5 6) false)
(check-expect (consecutive-num? 6 3 4) false)

;; two-same-num? Num Num Num -> Boolean
;; Purpose:
;; Produce true if (at least) two of the numbers are the same
;; and false otherwise
;; Examples:
(check-expect (two-same-num? 1 1 3) true)
(check-expect (two-same-num? 1 2 3) false)
;; Definition
(define (two-same-num? num1 num2 num3)
  (or (= num1 num2) (= num1 num3) (= num2 num3)))
;; Tests:
(check-expect (two-same-num? 1 1 1) true)
(check-expect (two-same-num? 1 3 1) true)
(check-expect (two-same-num? 3 1 1) true)

;; hand-value: Card Card Card -> Symbol
;; Purpose:
;; Given 3 cards, output the best hand value
;; straight-flush, flush, straight, three-of-a-kind, pair, high-card
;; Examples:
(check-expect (hand-value card1 card2 card3) 'straight-flush)
(check-expect (hand-value card1 card4 card5) 'three-of-a-kind)
;; Definition
(define (hand-value c1 c2 c3)
  (cond
    [(and (same-suit? c1 c2 c3) (consecutive-num? (card-rank c1) (card-rank c2)
(card-rank c3))) 'straight-flush]
    [(same-suit? c1 c2 c3) 'flush]
    [(consecutive-num? (card-rank c1) (card-rank c2) (card-rank c3)) 'straight]
    [(all-same-num? (card-rank c1) (card-rank c2) (card-rank c3)) 'three-of-a-kind]
    [(two-same-num? (card-rank c1) (card-rank c2) (card-rank c3)) 'pair]
    [else 'high-card]))
;; Tests
(check-expect (hand-value card1 card2 (make-card 'spades 13)) 'flush)
(check-expect (hand-value card1 card4 card3) 'pair)
(check-expect (hand-value card5 card3 card2) 'straight)
(check-expect (hand-value card1 card2 (make-card 'clubs 13)) 'high-card)
```