```
;; CS 135
;; Assignment 4
;; Winter 2014

;; Question 2

;; part (a)
;; sum-fav: (listof Int) Int -> Int
;; Purpose:
;; produce the total of all numbers in lst that
;; are greater than or eqaul to favourite
;; Examples:
(check-expect (sum-fav (cons 1 (cons 3 (cons 6 empty))) 3) 9)
(check-expect (sum-fav empty 4) 0)
(check-expect (sum-fav (cons 1 (cons 3 empty)) 10) 0)
(define (sum-fav lst favourite)
  (cond
    [(empty? lst) 0]
    [(>= (first lst) favourite) (+ (first lst)
                                   (sum-fav (rest lst) favourite))]
    [else (sum-fav (rest lst) favourite)]))
;; Tests:
(check-expect (sum-fav (cons -8 (cons -2 (cons -1 empty))) -3) -3)
(check-expect (sum-fav (cons -8 (cons -2 (cons -1 empty))) -5) -3)

;; part (b)
;; reciprocate: (listof Int) -> (listof (union Num 'undefined))
;; Purpose:
;; produce a list of all the reciprocal values in the list lst
;; Examples:
(check-expect (reciprocate (cons 1 (cons -2 (cons 0 empty))))
              (cons 1 (cons -1/2 (cons 'undefined empty))))
(check-expect (reciprocate empty) empty)
;; Definition
(define (reciprocate lst)
  (cond
    [(empty? lst) empty]
    [(zero? (first lst)) (cons 'undefined (reciprocate (rest lst)))]
    [else (cons (/ 1 (first lst)) (reciprocate (rest lst)))]))
;; Tests
(check-expect (reciprocate (cons 16 (cons 3 (cons -8 (cons 0 empty)))))
              (cons 1/16 (cons 1/3 (cons -1/8 (cons 'undefined empty)))))

;; part (c)
;; ascending-or-descending?: (listof Int) Symbol -> Boolean
;; Purpose:
;; produces true if the given list lst is either ascending or
;; descending, and the symbol direction matches (i.e., is
;; 'ascending or 'descending respectively)
;; Examples:
(check-expect
 (ascending-or-descending? (cons 1 (cons 2 (cons 4 empty))) 'ascending)
 true)
(check-expect
 (ascending-or-descending? (cons -1 (cons 2 (cons -4 empty))) 'descending)
 false)
(check-expect
 (ascending-or-descending? empty 'ascending) true)
```

```
;; Definition
(define (ascending-or-descending? lst direction)
  (cond
    [(empty? lst) true]
    [(empty? (rest lst)) true]
    [(symbol=? 'ascending direction)
     (and (< (first lst) (first (rest lst)))
          (ascending-or-descending? (rest lst) direction))]
    [(symbol=? 'descending direction)
     (and (> (first lst) (first (rest lst)))
          (ascending-or-descending? (rest lst) direction))]))
;; Tests:
(check-expect
 (ascending-or-descending? (cons 1 (cons 4 (cons 3 empty))) 'descending)
 false)
(check-expect
 (ascending-or-descending? (cons 1 (cons 4 empty)) 'descending) false)
(check-expect
 (ascending-or-descending? (cons 1 (cons 0 (cons -14 empty))) 'descending)
 true)



;; Question 3

;; part (a)

(define-struct line (slope intercept))
;; A Line=(make-line Num (union Num Symbol))

;; Template
;; my-linelst-fn: (listof Line) -> Any
(define (my-linelst-fn alinelst)
  (cond
    [(empty? alinelst) ...]
    [else
     (... (line-slope (first alinelst)) ...
      ... (line-intercept (first alinelst)) ...
      ... (my-linelst-fn (rest alinelst)) ...)]))


;; Some constants used for testing
(define line1 (make-line 1 0))
(define line2 (make-line 0 3))
(define line3 (make-line -1/2 -2))
(define line4 (make-line 'undefined -4))
(define line-list (cons line1 (cons line2 (cons line3 (cons line4 empty)))))


;; part (b)
;; negate-slope: (listof Line) -> (listof Line)
;; Purpose:
;; produces a list containing all lines in linelst with all
;; slopes negated (i.e., multiplied by -1)
;; Examples:
(check-expect (negate-slope empty) empty)
(check-expect (negate-slope line-list)
              (cons (make-line -1 0)
```

```
                    (cons line2
                          (cons (make-line 1/2 -2)
                                (cons line4 empty)))))
;; Definition:
(define (negate-slope linelst)
  (cond
    [(empty? linelst) empty]
    [(number? (line-slope (first linelst)))
     (cons (make-line (- (line-slope (first linelst)))
                      (line-intercept (first linelst)))
           (negate-slope (rest linelst)))]
    [else (cons (first linelst) (negate-slope (rest linelst)))]))
;; Tests:
;; above tested:  empty, -, +, 0 and 'undefined

;; part (c)
;; positive-line: (listof Line) -> (listof Line)
;; Purpose:
;; produces a list of all those lines from linelst which have
;; positive slope or a positive intercept
;; Examples:
(check-expect (positive-line line-list)
              (cons line1 (cons line2 empty)))
(check-expect (positive-line empty) empty)
;; Definition:
(define (positive-line linelst)
  (cond
    [(empty? linelst) empty]
    [(or (and (number? (line-slope (first linelst)))
          (> (line-slope (first linelst)) 0))
         (> (line-intercept (first linelst)) 0))
     (cons (first linelst) (positive-line (rest linelst)))]
    [else (positive-line (rest linelst))]))
;; Tests:
(check-expect
 (positive-line
  (cons (make-line 0 0)
        (cons (make-line -1 1)
              (cons (make-line 1 -1)
                    (cons (make-line -1 0)
                          (cons (make-line 0 -1)
                                (cons (make-line 1 1) empty)))))))
 (cons (make-line -1 1)
       (cons (make-line 1 -1)
             (cons (make-line 1 1) empty))))
(check-expect
 (positive-line
  (cons (make-line 'undefined 0)
        (cons (make-line 'undefined -1)
              (cons (make-line 'undefined 4) empty))))
 (cons (make-line 'undefined 4) empty))

;; part (d):

;; check-point?: Line Posn -> Boolean
;; Purpose:
;; produces true if the point pt in on line L, and false otherwise
;; Examples:
```

```
(check-expect (check-point? (make-line 2 0) (make-posn 1 2)) true)
(check-expect (check-point? (make-line 'undefined 5) (make-posn 5 13)) true)
;; Definition:
(define (check-point? L pt)
  (cond
    [(number? (line-slope L))
     (= (posn-y pt) (+ (* (line-slope L) (posn-x pt)) (line-intercept L)))]
    [else (= (posn-x pt) (line-intercept L))]))

;; Tests:
(check-expect (check-point? (make-line 2 0) (make-posn 4 5)) false)
(check-expect (check-point? (make-line 0 7) (make-posn 8 7)) true)
(check-expect (check-point? (make-line 0 7) (make-posn 8 9)) false)

;; through-point: (listof Line) Posn -> (listof Line)
;; Purpose:
;; produce a list of all lines from linelst which go through the point pt
;; Examples:
(check-expect
 (through-point line-list (make-posn -4 3))
 (cons (make-line 0 3) (cons (make-line 'undefined -4) empty)))
(check-expect (through-point empty (make-posn 3 6)) empty)
;; Definition
(define (through-point linelst pt)
  (cond
    [(empty? linelst) empty]
    [(check-point? (first linelst) pt)
     (cons (first linelst) (through-point (rest linelst) pt))]
    [else (through-point (rest linelst) pt)]))
;; Tests:
(check-expect
 (through-point (cons (make-line 0 0)
                      (cons (make-line 'undefined 0)
                            (cons (make-line -71 0) empty))) (make-posn 0 0))
 (cons (make-line 0 0)
                      (cons (make-line 'undefined 0)
                            (cons (make-line -71 0) empty)))))

;; part (e):
;; parallel-non-intersect: (listof Line) -> (listof Boolean)
;; Purpose:
;; produce a list of boolean values representing which consecutive pairs
;; of lines in linelst are parallel and non-intersecting
;; Examples:
(check-expect (parallel-non-intersect (cons (make-line 3 4) (cons (make-line 3 8)
empty)))
                                      (cons true empty))
(check-expect (parallel-non-intersect (cons (make-line 3 4) (cons (make-line 3 4)
empty)))
                                      (cons false empty))
(check-expect (parallel-non-intersect (cons (make-line 3 4) (cons (make-line 3 4)
empty)))
                                      (cons false empty))
;; Definition:
(define (parallel-non-intersect linelst)
  (cond
    [(or (empty? linelst) (empty? (rest linelst))) empty] ;; base cases: 0 or 1 item
    ;; we have at least two elements
```

```
    [else
     (cons
      (and
       (equal? (line-slope (first linelst)) (line-slope (first (rest linelst))))
       (not (equal? (line-intercept (first linelst)) (line-intercept (first (rest
linelst))))))
      (parallel-non-intersect (rest linelst)))]))
;; Tests:
(check-expect
 (parallel-non-intersect
  (cons (make-line 0 5) (cons (make-line 0 8) (cons (make-line 8 0) (cons
(make-line 5 4) empty)))))
 (cons true (cons false (cons false empty))))
(check-expect
 (parallel-non-intersect
  (cons (make-line 0 8) (cons (make-line 'undefined 8) (cons (make-line 'undefined
0) (cons (make-line 5 0) empty)))))
 (cons false (cons true (cons false empty))))
```