```
;; Question 2:

(define-struct node (key val left right))

;; bst-remove : Bst Num -> Bst
;; Removes the node with the given key from the given binary search tree
;; Examples:
(check-expect (bst-remove empty 1) empty)
(check-expect (bst-remove (make-node 1 "" empty empty) 1) empty)
(check-expect (bst-remove (make-node 2 "" empty empty) 1)
              (make-node 2 "" empty empty))

(define (bst-remove t k)
  (cond
    [(empty? t) empty]
    [(< k (node-key t)) (make-node (node-key t) (node-val t)
                                   (bst-remove (node-left t) k)
                                   (node-right t))]
    [(> k (node-key t)) (make-node (node-key t) (node-val t)
                                   (node-left t)
                                   (bst-remove (node-right t) k))]
    [(and (empty? (node-left t)) (empty? (node-right t))) empty]
    [(empty? (node-left t)) (node-right t)]
    [(empty? (node-right t)) (node-left t)]
    [else
     (local [
             ;; leftmost : BST[non-empty] -> node
             ;; Produces the leftmost node in the given BST
             (define (leftmost t)
               (cond [(empty? (node-left t)) t]
                     [else (leftmost (node-left t))]))

             ;; inorder-successor : BST[non-empty] -> node
             ;; Produces the inorder successor of the root of the given BST
             (define (inorder-successor t)
               (leftmost (node-right t)))

             (define the-successor (inorder-successor t))]
       (make-node (node-key the-successor) (node-val the-successor)
                  (node-left t)
                  (bst-remove (node-right t) (node-key the-successor))))]))

;; Tests:
(check-expect (bst-remove (make-node 2 "" (make-node 1 "" empty empty) empty) 2)
              (make-node 1 "" empty empty))
(check-expect (bst-remove (make-node 2 "" empty (make-node 3 "" empty empty)) 2)
              (make-node 3 "" empty empty))
(check-expect (bst-remove (make-node 2 "" (make-node 1 "" empty empty)
                                          (make-node 3 "" empty empty)) 1)
              (make-node 2 "" empty (make-node 3 "" empty empty)))
(check-expect (bst-remove (make-node 2 "" (make-node 1 "" empty empty)
                                          (make-node 3 "" empty empty)) 2)
              (make-node 3 "" (make-node 1 "" empty empty) empty))
(check-expect (bst-remove (make-node 2 "" (make-node 1 "" empty empty)
                                          (make-node 3 "" empty empty)) 3)
              (make-node 2 "" (make-node 1 "" empty empty) empty))
(check-expect (bst-remove (make-node 2 "" (make-node 1 "" empty empty)
                                          (make-node 4 "" (make-node 3 "" empty empty)
```

```
empty)) 2)
              (make-node 3 "" (make-node 1 "" empty empty) (make-node 4 "" empty
empty)))

;; Question 3
;; part (a)
;; lookup-al: AL Symbol -> (union false Num)
;; Purpose:
;; produces the key assocatied with key in the association list al,
;; or false, if the key does not appear in al
;; Examples:
(check-expect (lookup-al (list (list 'a 5) (list 'b 3)) 'b) 3)
(check-expect (lookup-al (list (list 'a 5) (list 'b 3)) 'd) false)
(define (lookup-al al key)
  (cond
    [(empty? al) false]
    [(symbol=? (first (first al)) key) (second (first al))]
    [else (lookup-al (rest al) key)]))
;; Tests
(check-expect (lookup-al '((x 2) (y 3)) 'x) 2)
(check-expect (lookup-al '((x 2) (y 3)) 'y) 3)
(check-expect (lookup-al '((x 2) (y 3)) 'z) false)


;; part (b)
;; eval1: AE AL -> Num
;; Purpose:  produces the value of the given arithmetic expression exp
;; using the bindings of variables to values given in vars
;; Examples:
(check-expect (eval1 '(+ x (* y 2)) '((x 2) (y 3) (z 4))) 8)
(check-expect (eval1 '(+ 1 2) empty) 3)
(define (eval1 exp vars)
  (cond [(number? exp) exp]
        [(symbol? exp) (lookup-al vars exp)]
        [else (apply1 (first exp) (rest exp) vars)]))
;; Tests:
(check-expect (eval1 '(+ x (* x 2)) '((x 2) (y 3) (z 4))) 6)
(check-expect (eval1 '(+ x (* z 2)) '((x 2) (y 3) (z 4))) 10)
(check-expect (eval1 '(* x (+ y 2)) '((x 2) (y 3) (z 4))) 10)
(check-expect (eval1 'x '((x 3) (y 8))) 3)

;; apply1: Symbol AEList AL -> Num
;; Purpose:
;; produce the number resulting from applying the function f
;; to each of the arithmetic expressions in aaxl using the
;; bindings of variables to values given in the association list
;; vars
;; Examples:
(check-expect (apply1 '+ '(x (* y 2)) '((x 2) (y 3) (z 4))) 8)
(check-expect (apply1 '+ '(3 5) empty) 8)
(define
  (apply1 f aaxl vars)
  (cond
    [(and (empty? aaxl) (symbol=? f '*))  1]
    [(and (empty? aaxl) (symbol=? f '+))  0]
    [(symbol=? f '*)
     (* (eval1 (first aaxl) vars)
        (apply1  f (rest aaxl) vars))]
```

```
    [(symbol=? f '+)
     (+ (eval1 (first aaxl) vars)
        (apply1 f (rest aaxl) vars))]))
;; Tests
(check-expect (apply1 '+ '(x (* x 2)) '((x 2) (y 3) (z 4))) 6)
(check-expect (apply1 '+ '(x (* z 2)) '((x 2) (y 3) (z 4))) 10)
(check-expect (apply1 '* '(x (+ y 2)) '((x 2) (y 3) (z 4))) 10)


;; part (c)
;; eval2: LODOE -> Num
;; Purpose:  produces the value of the given arithmetic expression exp
;; using the bindings of variables to values given in vars
;; Examples:
(check-expect (eval2 '((define x 2) (define y 3) (define z 4) (+ x (* y 2)))) 8)
(check-expect (eval2 '((define x 2) (+ 1 2))) 3)
(define (eval2 ae)
  (local
    [(define (e-help ae env)
       (cond
         [(number? (first ae)) (first ae)]
         [(symbol? (first ae))
          (lookup-al env (first ae))]
         [(symbol=? (first (first ae)) 'define)
          (e-help (rest ae)
                  (cons (list
                         (second (first ae))
                         (third (first ae)))
                        env))]
         [else
          (apply2 (first (first ae))
                  (rest (first ae)) env)]))]
    (e-help ae '())))
;; Tests
(check-expect (eval2 '((+ 1 2))) 3)
(check-expect (eval2 '((define x 2) (+ x 2))) 4)
(check-expect (eval2 '((define x 2) x)) 2)
(check-expect (eval2 '((define x 2) (define y 8) 14)) 14)
(check-expect (eval2 '((define x 2) (define y 8) (+ y x))) 10)

;; apply2: Symbol AEList AL -> Num
;; Purpose:
;; produce the number resulting from applying the function f
;; to each of the arithmetic expressions in aaxl using the
;; bindings of variables to values given in the association list
;; vars
;; Examples:
(check-expect (apply2 '+ '(x (* y 2)) '((x 2) (y 3) (z 4))) 8)
(check-expect (apply2 '+ '(3 5) empty) 8)
(define (apply2 fn aelst env)
  (local
    [(define (eval2 ae env)
       (cond
         [(number? ae)  ae]
         [(symbol? ae) (lookup-al env ae)]
         [else (apply2 (first ae) (rest ae) env)]))]
    (cond
      [(and (empty? aelst) (symbol=? fn '+)) 0]
```

```
      [(and (empty? aelst) (symbol=? fn '*)) 1]
      [(symbol=? fn '+)
       (+ (eval2 (first aelst) env)
          (apply2 fn (rest  aelst) env))]
      [else
       (* (eval2 (first aelst) env)
        (apply2 fn (rest aelst) env))])))
;; Tests
(check-expect (apply2 '+ '(x (* y 2)) '((x 2) (y 3) (z 4))) 8)
(check-expect (apply2 '+ '(3 5) empty) 8)
(check-expect (apply2 '+ '(x (* x 2)) '((x 2) (y 3) (z 4))) 6)
(check-expect (apply2 '+ '(x (* z 2)) '((x 2) (y 3) (z 4))) 10)
(check-expect (apply2 '* '(x (+ y 2)) '((x 2) (y 3) (z 4))) 10)


;; Question 4:
;; mapfn: (listof (Num Num -> Any)) (list Num Num) -> (listof Any)
;; also OK to replace both of the above "Any" by "X"
;; Purpose:
;; Produce the list of results of applying each of the function in lof
;; to the given two parameters in loparms
;; Examples:
(check-expect (mapfn empty '(0 0)) empty)
(check-expect (mapfn (list + - * / list) '(3 2)) '(5 1 6 1.5 (3 2)))
(define (mapfn lof loparms)
  (cond
    [(empty? lof) empty]
    [else (local
            [(define f (first lof))]
            (cons (f (first loparms) (second loparms))
                (mapfn (rest lof) loparms)))]))
;; Tests:
(check-expect (mapfn (list list) '(0 9)) (list (list 0 9)))
(check-expect (mapfn (list expt +) '(4 3)) (list 64 7))
```