```
;; -------------
;; Question 1


(define x 1)
(define y 2)

(+ x
   (local [(define x 2)
           (define (foo y)
             (* x y))]
     (foo 3)))

;(A)


;(B) note y should not be renamed

(define x_0 2)
(define (foo_0 y)
  (* x_0 y))

;(C)

(+ 1 (* x_0 3))



;; -------------
;; Question 2

(define-struct unit (name subs))
;;
;; A Unit is (make-unit n subs) where
;;   n is String
;;   subs is (listof Unit)
;;

;; 2A
;;
;; ListOfUnit is one of:
;;  - empty
;;  - (cons u lou) where
;;    u is Unit
;;    lou is ListOfUnit

;; 2B
(make-unit "a" empty)

(make-unit "a" (list (make-unit "b" empty)
                     (make-unit "c" (list (make-unit "d" empty)))))
```

```
;2C
Unit:            binary tree                    (n-ary tree)
Unit:            fixed size                     (arbitrary size)
(listof Unit):   fixed size                     (arbitrary size)
(listof Unit):  (involved in mutual-reference)   self reference
Unit:           (single case)                    multiple cases
```

```
;; 2D

(define (fun-for-unit u)
  (... (unit-name u)
       (fun-for-lou (unit-subs u))))      ;MR
```

```
(define (fun-for-lou lou)
  (cond [(empty? lou) ...]
        [else
         (... (fun-for-unit (first lou))  ;MR
              (fun-for-lou (rest lou)))]));NR



;; 2E
;; count-units: Unit -> Number
;; produce count of how many units u comprises, including itself and all subs
(check-expect (count-units (make-unit "a" empty)) 1)

(check-expect (count-units
               (make-unit "b" (list (make-unit "b" empty)
                                    (make-unit "c" (list (make-unit "d" empty))))))
              4)

;; does NOT have to be written w/ local
(define (count-units u)
  (local [(define (fun-for-unit u)
            (+ 1
               (fun-for-lou (unit-subs u))))

          (define (fun-for-lou lou)
            (cond [(empty? lou) 0]
                  [else
                   (+ (fun-for-unit (first lou))
                      (fun-for-lou (rest lou)))]))]
    (fun-for-unit u)))


;; ----------
;; Question 3
;;

;; intersection: (listof Number) (listof Number) -> (listof Number)
;; produces list containing all numbers in l1 that also appear in l2
(check-expect (intersection empty      empty)      empty)
(check-expect (intersection empty      (list 3 4)) empty)
(check-expect (intersection (list 1 2) (list 3 2)) (list 2))

(define (intersection l1 l2)
  (cond [(empty? l1) empty]
        [else
         (if (is-in? (first l1) l2)
             (cons (first l1) (intersection (rest l1) l2))
             (intersection (rest l1) l2))]))

#; ;; OR
(define (intersection l1 l2)
  (local [(define (in-l2? n)
            (is-in? n l2))]
    (filter in-l2? l1)))


;; is-in? Number (listof Number) -> Boolean
;; produces if n appears in lon
(check-expect (is-in? 1 empty) false)
(check-expect (is-in? 4 (list 3 4)) true)
(check-expect (is-in? 1 (list 3 2)) false)

(define (is-in? n lon)
  (cond [(empty? lon) false]
        [else (or (= n (first lon))
                  (is-in? n (rest lon)))]))
```

```
;; ----------
;; Question 4


;; squares: (listof Number) -> (listof Number)
(check-expect (squares (list 2 3 4)) (list 4 9 16))
#;
(define (squares lon)
  (cond [(empty? lon) empty]
        [else
         (cons (sqr (first lon))
               (squares (rest lon)))]))

(define (squares lon) (map sqr lon))




;; powers: Number (listof Number) -> (listof Number)
(check-expect (powers 3 (list 2 3 4)) (list 8 27 64))
#;
(define (powers n lon)
  (cond [(empty? lon) empty]
        [else
         (cons (expt (first lon) n)
               (powers n (rest lon)))]))

(define (powers n lon) (local [(define (p x) (expt x n))] (map p lon)))




;; just-within: (listof Posn) -> (listof Posn)
(check-expect (just-within (list (make-posn 1 3)
                                 (make-posn -1 3)
                                 (make-posn 1 -3)))
              (list (make-posn 1 3)))
#;
(define (just-within lop)
  (cond [(empty? lop) empty]
        [else
         (if (within? (first lop))
             (cons (first lop) (just-within (rest lop)))
             (just-within (rest lop)))]))

(define (just-within lop) (filter within? lop))

(define (within? p)
  (and (< 0 (posn-x p) 4)
       (< 0 (posn-y p) 4)))



;; digits->num: (listof Digit) -> Integer
(check-expect (digits->num empty) 0)
(check-expect (digits->num (list 2)) 2)
(check-expect (digits->num (list 3 2)) 23)
(check-expect (digits->num (list 3 2 4)) 423)
#;
(define (digits->num lod)
  (cond [(empty? lod) 0]
        [else
         (+ (first lod)
            (* 10 (digits->num (rest lod))))]))


(define (digits->num lod)
  (local [(define (combine f r) (+ f (* 10 r)))]
    (foldr combine 0 lod)))


;; ----------
;; Question 5

;(define-struct unit (name subs))
;;
;; A Unit is (make-unit n subs) where
;;   n is String
;;   subs is (listof Unit)
(check-expect (equal-units? (make-unit "foo" empty)
                            (make-unit "foo" empty))
              true)
(check-expect (equal-units? (make-unit "foo" empty)
                            (make-unit "bar" empty))
              false)
(check-expect (equal-units? (make-unit "foo" (list (make-unit "a" (list (make-unit "b"
empty)))))
                            (make-unit "foo" (list (make-unit "a" (list (make-unit "b"
empty))))))
              true)

(define (equal-units? u1 u2)
  (local [(define (equal-units? u1 u2)
            (and (string=? (unit-name u1) (unit-name u2))
                 (equal-lou? (unit-subs u1) (unit-subs u2))))
          (define (equal-lou? lou1 lou2)
            (cond [(and (empty? lou1) (empty? lou2)) true]
                  [(and (empty? lou1) (cons?  lou2)) false]    ;<<<<
                  [(and (cons?  lou1) (empty? lou2)) false]    ;<<<<
                  [else
                   (and (equal-units? (first lou1) (first lou2))
                        (equal-lou? (rest lou1)
                                    (rest lou2)))]))]
    (equal-units? u1 u2)))
```

```
;
 ;; start with:
 [(and (empty? lou1) (cons?  lou2)) false]
 [(and (cons?  lou1) (empty? lou2)) false]


 ;; combine same answers
 [(or (and (empty? lou1) (cons?  lou2))
      (and (cons?  lou1) (empty? lou2)))   false]

 ;; exploit order, previous tests is already (and empty? empty?)
 [(or (empty? lou1) (empty? lou2)) false]
```

```
;; ----------
;; Extra Credit

(check-expect (ndigits->num empty)          0)
(check-expect (ndigits->num     (list 2))   2)
(check-expect (ndigits->num   (list 3 2))  32)
(check-expect (ndigits->num (list 1 3 2)) 132)

#;
(define (ndigits->num lod)
  (local [(define (d->n acc lod)
            (cond [(empty? lod) acc]
                  [else
```

```
                   (d->n (+ (* 10 acc) (first lod)) (rest lod))]))]
    (d->n 0 lod)))


(define (ndigits->num lod)
  (local [(define (combine d acc) (+ (* 10 acc) d))]
    (foldl combine 0 lod)))
```