```
;;
;; CPSC 110, Fall 2010, Midterm 2
;;
;;

;; Lexical Scoping [5 points]
;;
;; There are 5 variable REFERENCES in this short program. Draw
;; arrows from each reference to the parameter or define that
;; provides the value for it.  (We are looking for arrows like
;; those that the DrRacket "check syntax" button enables.
;;


;[1 POINT FOR EACH OF 5 REFERENCES]
;[if functions are treated as variables correctly give points]
;[but take off 1 for each incorrect arrow]
#;#;#;
(define A:a 1)

(define (foo B:b)

  (local [(define (bar C:c)

            (* a{D} b{B} c{C}))

          (define D:a 10)]

    (bar a{D})))

(foo a{A})


;; Evaluation of Local [10 Points]
;; The following program has a definition followed by an expression.

(define (f1 n lon)
  (local [(define (f2 m) (* n m))]
    (map f2 lon)))

(f1 3 (list 2 4 6))

;; What is the value of the expression?
(list 6 12 18) ;[4 points]


;; Show the state of the hand execution, including lifted definitions,
;; right before the call to the map primitive returns.

(define (f2_0 m) (* 3 m)) ;[2 pts for each of 2 renamings]
(map f2_0 (list 2 4 6))   ;[2 points for proper lifting]







;;-----------------------------------
;; Problem 3 [30 points]

;; Consider the following structure definitions and type comments:

(define-struct kvp (k v))
;; KVP is (make-kvp Integer String)
;; interp. a key/value pair
```

```
(define-struct node (p subs))
;; Tree is (make-node KVP[R] ListOfTree[MR])
;; interp. an arbitrary arity tree, each node has a KVP

;; ListOfTree is one of:
;; - empty
;; - (cons Tree[MR] ListOfTree[SR])

;; [ 2 points for each of 4 correct arrows]


;; Define TWO examples for EACH of the above THREE types.
;; Feel free to use variables you define for some examples
;; in the definition of other examples.

;; [1/2 point for each correct example, ROUND UP]

(define KVP1 (make-kvp 1 "a"))
(define KVP3 (make-kvp 3 "c"))

(define T1 (make-node KVP1 empty))
(define T3 (make-node KVP3 (list T2 (make-node (make-kvp 4 "d") empty))))

(define LOT1 empty)
(define LOT3 (node-subs T3))




;; NEATLY annotate the type comments above by drawing a line
;; from any type reference to the the corresponding type definition.
;; All your arrows should end at one of the names that appear right
;; before 'is'.
;; NEATLY label each line with one of MR, SR or R, depending on
;; whether the reference is a mutual reference, self reference
;; or an ordinary reference.




;; Design the function sum-keys, which consumes a Tree and produces
;; the sum of the keys in every KVP in the tree. When designing the
;; function you do not have to leave your template behind, but using
;; the redux page provided please write down, in order, the names of
;; the redux rules you used to generate the template.

;; Tree -> Integer               [1 points]
;; sum all the keys in t         [1 points]
(check-expect (sum-keys T1) 1)  ;[1 point for base first]
(check-expect (sum-keys T2) 3)  ;
(check-expect (sum-keys T3) 10) ;[1 point for 2 deep test]

;;[4 points, .5 point each, round up]
;;
;; compound          [Tree is (make-node )]
;; reference         [to KVP]
;; mutual reference  [to ListOfTree]
;; itemization       [ListOfTree is one of]
;; atomic distinct   [empty]
;; compound          [cons]
;; mutual reference  [Tree]
;; self-reference    [ListOfTree]

;; [3 points correct code, use of local NOT required]

(define (sum-keys t)
  (local [(define (sum-keys-tree t)
```

```
            (+ (kvp-k (node-p t))
               (sum-keys-lot (node-subs t))))

          (define (sum-keys-lot lot)
            (cond [(empty? lot) 0]
                  [else
                   (+ (sum-keys-tree (first lot))
                      (sum-keys-lot  (rest lot)))]))]
    (sum-keys-tree t)))

;; Now consider the design of the function sum-val-lengths, which
;; consumes a Tree and produces the sum of the string lengths of
;; the keys. Will the design of sum-val-lengths have the same
;; number of functions as sum-keys? Why or why not?

;; [3 points, 1 for 1 more, 2 for right reasoning]
;; It will have one more because when sum-val-lengths needs the val
;; length for a key it must call a helper function. In this case
;; the function will not already exist, it will have to be designed.

;; [the code below is NOT necessary]
;; Tree -> Natural
;; sum all the key lengths in t
(check-expect (sum-val-lengths T1) 1)
(check-expect (sum-val-lengths T2) 2)
(check-expect (sum-val-lengths T3) 4)

(define (sum-val-lengths t)
  (local [(define (sum-val-lengths-tree t)
            (+ (sum-val-lengths-kvp (node-p t))
               (sum-val-lengths-lot (node-subs t))))

          (define (sum-val-lengths-kvp kvp)
            (string-length (kvp-v kvp)))

          (define (sum-val-lengths-lot lot)
            (cond [(empty? lot) 0]
                  [else
                   (+ (sum-val-lengths-tree (first lot))
                      (sum-val-lengths-lot  (rest lot)))]))]
    (sum-val-lengths-tree t)))



;; -----------------------------------------------------
;; Problem 4 [25 Points]
;;
;; In this problem you will design a function to merge two
;; lists of numbers that are already sorted in ascending order
;; into a single list of numbers, also sorted in ascending
;; order. Call the function merge. For example:

;;
;;  (merge (list 3 6) (list 1 4 8)) produces  (list 1 3 4 6 8)
;;
;; In addition to the usual signature, purpose and examples,
;; your solution should show the step by step process going
;; from the first version of the function, where the template
;; has been filled in to get code that correctly passes the
;; tests, to the final version after simplification.

;; [first grade the original solution
;; [2 points for signature
;; [1 point each for 4 tests that are cross-product of cases
;; [1 point each for 4 cond questions
;; [1 point each for 4 correct answers
;;
```

```
;; [ if original solution is simplified rather than starting w
;; [ all four cases the 8 points for questions and answers above
;; [ are not awarded
;; [
;; [then assign up to 11 more points as follows for simplified
;; [solutions:
;; [  - just else 3 points
;; [  - case reduction 5 points
;; [  - as simple as below 8 points
;; [  - each sound explanation 3 points
;;


;; LON LON -> LON
(check-expect (merge empty empty) empty)
(check-expect (merge empty (list 2 3)) (list 2 3))
(check-expect (merge (list 1 2) empty) (list 1 2))
(check-expect (merge (list 3 6) (list 1 4 8)) (list 1 3 4 6 8))
#;
(define (merge l1 l2)
  (cond [(and (empty? l1) (empty? l2)) empty]
        [(and (empty? l1) (cons? l2)) l2]
        [(and (cons?  l1) (empty? l2)) l1]
        [(and (cons? l1) (cons? l2))
         (if (< (first l1) (first l2))
             (cons (first l1) (merge (rest l1) l2))
             (cons (first l2) (merge l1 (rest l2))))]))

;;
;; Given the first case, which tests that both l1 and l2 are
;; empty, the two second tests are effectively:
;;
;;  (and (empty? l1) true)  since l2 can't be empty if l1 is
;;  (and true (empty? l2))  since l1 can't be empty if l2 is
;;
;; which simplify to:
;;
;;  (empty? l1)
;;  (empty? l2)
#;
(define (merge l1 l2)
  (cond [(and (empty? l1) (empty? l2)) empty]
        [(empty? l1)  l2]
        [(empty? l2) l1]
        [(and (cons? l1) (cons? l2))
         (if (< (first l1) (first l2))
             (cons (first l1) (merge (rest l1) l2))
             (cons (first l2) (merge l1 (rest l2))))]))
;;
;; the first two cases are now effectively the same
;; so the first can be deleted, and the third question
;; can be replaced w/ else
;;

(define (merge l1 l2)
  (cond [(empty? l1) l2]
        [(empty? l2) l1]
        [else
         (if (< (first l1) (first l2))
             (cons (first l1) (merge (rest l1) l2))
             (cons (first l2) (merge l1 (rest l2))))]))


;; -----------------------------------------------------
;; Problem 5 [20 points]
;; Below are two functions excerpted from a version of the fireworks world program.
;; Improve the code for these functions using the built in abstract list functions
```

```
;; listed in the appendix to this midterm.
;;
;; You only need to rewrite the body of each function, the signature, purpose and
;; examples/tests are already there for you. You should only need to use one abstract
;; list function in each of these functions.
```

```
;; (listof Firework) -> (listof Firework)
(check-expect (tick-fws empty) empty)
(check-expect (tick-fws (list (make-fw 10 300 "red" 20)
                              (make-fw 10 100 "blue" 0)))
              (list (make-fw 10 (- 300 SPEED) "red" 19)
                    (make-fw 10 (- 100 SPEED) "blue" -1)))

(define (tick-fws lofw)
  (cond [(empty? lofw) empty]
        [else
         (cons (tick-fw (first lofw))
               (tick-fws (rest lofw)))]))

;; [all 10 points for correct code]
;; [- 3 for serious syntax errors like (map (tick-fw fw) lofw)
;; [4 points for presence of map, but rest is far off

(define (tick-fw lofw)
  (map tick-fw lofw))
```

```
;; (listof Firework) -> Image
;; render all fireworks in lofw onto MTS
(check-expect (render-fws empty) BG)
(check-expect (render-fws (list (make-fw 10 20 "red" 1)
                                (make-fw 30 40 "blue" 0)))
              (place-image (circle RADIUS "solid" "red") 10 20
                           (place-image (radial-star 8 8 30 "solid" "blue") 30 40 BG)))

(define (render-fws lofw)
  (cond [(empty? lofw) BG]
        [else
         (place-fw (first lofw)
                   (render-fws (rest lofw)))]))


;; [all 10 points for correct code]
;; [- 3 for serious syntax errors like (foldr (place-fw fw) BG lofw)
;; [4 points for presence of foldr, but rest is far off
;; [foldl is ok

(define (render-fws lofw)
  (foldr place-fw BG lofw))
```

```
;; -----------------------------------------------------
;;
;; Designing abstract functions. [10 points]

;; The template for a function operating on (listof X) is
;;
;;  (define (fn-for-lox lox)
;;    (cond [(empty? lox) (...)]
;;          [else
;;            (... (first lox)
;;                 (fn-for-lox (rest lox)))]))
;;
```

```
;; The template for a function operating on Natural is
;;
;;  (define (fn-for-nat n)
;;    (cond [(zero? n) (...)]
;;          [else
;;            (... n
;;                 (fn-for-nat (sub1 n)))]))
;;
;; The abstract functions based on these templates are called fold and natfun.
;;
;; Using fold, it is easy to write a function that consumes (listof Number)
;; lon and produces the sum of the numbers in lon.

;; (listof Number) -> Number
;; produce sum of elements of lon
(check-expect (sumlon (list 3 2 1 0)) 6)

(define (sumlon lon) (fold + 0 lon))


;; Using natfun it is easy to write a function that consumes Natural n and
;; produces the sum of the n natural numbers.

;; Natural -> Natural
;; produce sum of first n natural numbers
(check-expect (sumnat 3) 6)

(define (sumnat n) (natfun + 0 n))

;;
;; Consider designing a function called red that abstracts both fold and
;; natfun. Such a function could be used to write both sumlon and sumnat.
;;
;; How many parameters would the abstract function have?

;;  FIVE

;;
;; Write the definitions of sumlon and sumnat assuming the abstract
;; function exists. You do not have to write the abstract function,
;; but you may find it helpful to do so. Call it red.

;;
;; [5/6 + correct code  -> 10
;; [5   + rest is wrong -> 4
;; [6   + rest is wrong -> 3
;; [4   + correct except missing fn arg -> 8
;; [4   + complete correct code that just calls fold/natfun -> 5
;; [    code that passes 2 or more of empty?/rest -> 7
;; [    code that passes 1 or more of empty?/rest -> 3
;;
;;


#;#;
(define (sumlon lon)
  (local [(define (comb lon nr) (+ (first lon) nr))]
    (red comb empty? rest 0 lon)))

(define (sumnat n)
  (red + zero? sub1 0 n))

(define (red fn base? next b v)
  (cond [(base? v) b]
        [else
         (fn v (red fn base? next b (next v)))]))
```

```
(define (fold fn b lox)
  (local [(define (comb lox nrresult)
            (fn (first lox) nrresult))]
    (red comb empty? rest b lox)))

(define (natfun fn b n)
  (red fn zero? sub1 b n))
```