Q1. Given the following function definition:

(define (bar x y)
  (local [((define (foo v) (* v y))]
    (* (foo x) y))))

a) What is the value of the following expression? Note that it is not necessary to show the step-by-step evaluation, although you may want to use the spare page provided at the start of this exam to sketch it out.

(+ (bar 3 1)
  (bar 4 2))

Answer: 19   [1 point]

b) How many function definitions are lifted during the evaluation of the expression given in part a)?

Answer: 2   [1 point]

c) Write out the lifted function definition(s)

Answer:

(define (foo_0 v) (* v 1))
(define (foo_1 v) (* v 2))

[1 point - proper renaming]
[1 point - parameter not changed]
[1 point - closed over parameter changed correctly]
Q2. Given the following partial data definitions:

```
(define-struct bag (l w h))
;; Bag is (make-bag Number Number Number)
;; interp. a bag with a length, width and height in inches
(define B1 (make-bag 19.5 10.0 6.5))
(define B2 (make-bag 23.0 11.5 7.0))
(define B3 (make-bag 18.0 9.5 5.5))

;; ListOfBag is one of:
;; - empty
;; - (cons Bag ListOfBag)
;; interp. a list of bags
(define LOB1 empty)
(define LOB2 (list B1 B2 B3))
```

The linear length of a bag is defined to be its length plus width plus height. Complete the design of the function `linear-length-lob` that consumes a list of bags and produces a list of the linear lengths of each of the bags in the list.

**Hint:** although it is not required, feel free to encapsulate any helper function(s) in a local expression so that you do not have to design tests for those functions.

**There is more space on the next page, if needed.**

```
;; ListOfBag -> ListOfNumber
;; produces list of linear lengths of all the bags in lob
(check-expect (linear-length-lob empty) empty)
(check-expect (linear-length-lob (list B1 B2 B3))
  (list 36.0 41.5 33.0))

;(define (linear-length-lob lob) empty) ;stub
```
Answer:

```
(define (linear-length-lob lob)
  (local [(define (linear-length b)
            (+ (bag-l b) (bag-w b) (bag-h b)))]
    (cond [(empty? lob) empty]
          [else
           (cons (linear-length (first lob))
                 (linear-length-lob (rest lob))))]))
```

[1 point - based on template for `ListOfBag`]
[1 point - correct base case result]
[1 point - correct combination]
[1 point - call to fn-for-bag]

Helper (need not be encapsulated in local):

If encapsulated in local:
[2 points - correct encapsulation
- [1 point] (local ...)
- [1 point] [(define (lin ...))]]

[1 point - based on template for `Bag`]
[1 point - correct function body]

otherwise
[1 point - correct signature and purpose]
[1 point - correct check-expect]
[1 point - based on template for `Bag`]
[1 point - correct function body]
Q3a) Use one or more built-in abstract functions to design a function \( \text{times2-lon} \) that consumes a list of numbers \( \text{lon} \) and produces a list consisting of each element of \( \text{lon} \) multiplied by 2. You can use a function encapsulated in a local expression or an anonymous function – your choice. You must preserve the template in your solution.

\[
\text{;; ListOfNumber -> ListOfNumber}
\text{;; produces a list of all the numbers in \text{lon} multiplied by 2}
\text{(check-expect \( \text{times2-lon} \) \((\text{list} \ 1 \ 2 \ 3)\) \((\text{list} \ 2 \ 4 \ 6)\))}
\]

\[(\text{define \( \text{times2-lon} \) \(\text{lon} \) empty)} \quad ;\text{stub}\]

**Answer:**

```
#; (define (times2-lon \text{lon})
   (map ... \text{lon}))

(define (times2-lon \text{lon})
   (map (\lambda (n) (* 2 n)) \text{lon}))
```

[2 points - template]
[2 points - function mapped onto \text{lon}:
- (\lambda (n) (* 2 n))

OR
- (local [(define (times2 n) (* 2 n))]
   (map times2 \text{lon}))]
```
b) Consider the following function definition that consumes a minimum size `min` and a list of dots `lod` and that produces a list consisting of all the elements of `lod` except for those whose size is smaller than `min`.

```scheme
;; Number ListOfDot -> ListOfDot
;; produces a list of dots in lod that are not smaller than min
(check-expect (remove-small 5 empty) empty)
(check-expect (remove-small 5 (list (make-dot 5 "blue")
    (make-dot 4 "red")
    (make-dot 6 "yellow")))
    (list (make-dot 5 "blue")
    (make-dot 6 "yellow"))

(define (remove-small min lod)
  (cond [(empty? lod) empty]
        [else (if (smaller? min (first lod))
                  (remove-small min (rest lod))
                  (cons (first lod)
                        (remove-small min (rest lod))))]))
```

You should assume that the function `smaller?` has been designed elsewhere in the code – you do not have to provide the design for this function.

Rewrite the function `remove-small` using a built-in abstract function – it is not necessary to rewrite the signature, purpose or check-expects. You can use a function encapsulated in a local expression or an anonymous function – your choice! You must preserve the template in your solution.

Answer:

```scheme
;;
(define (remove-small min lod)
  (filter (λ(d) (not (smaller? min d))) lod))
```

[2 points - function based on template for filter]
[3 points - function for filtering]
- (λ(d) (not (smaller? min d)))
-OR-
- (local [(define (not-smaller? d)
                      (not (smaller? min d)))]

- in each case:
  [2 points - (smaller? min d)]
  [1 point - (not (...))]
c) Use one or more built-in abstract functions to complete the design of a function that consumes a natural number n and produces an image of solid, red circles of radius 0, 1, 2, ..., n-1 beside one another. Read the provided check-expects carefully to ensure that you understand the role of the BLANK image. You can use a function encapsulated in a local expression or an anonymous function – your choice! For this question, it is not necessary to preserve the template in your solution.

```
(define BLANK (empty-scene 0 0))

;; Natural -> Image
;; produce an image of circles of radius 1, 2, ..., n-1 beside
;; one another
(check-expect (circles 0) BLANK)
(check-expect (circles 1) (beside
  (circle 0 "solid" "red"
  BLANK))
(check-expect (circles 3) (beside
  (circle 0 "solid" "red"
  (circle 1 "solid" "red"
  (circle 2 "solid" "red"
  BLANK))

(define (circles n) BLANK) ;stub
```

Answer:

```
(define (circles n)
  (foldr beside BLANK
    (map (λ(r) (circle r "solid" "red"))
      (build-list n identity)))))
```

[3 points] (foldr beside BLANK ...)  
- deduct (1) for each error

[3 points] (map (λ(r) (circle r "solid" "red")) ...)  
- OK if anonymous function defined in local expression  
- deduct (1) for each error

[3 points] (build-list n identity)  
- deduct (1) for each error]
Q4. Consider the following variable definitions and partial data definitions:

```
(define I0 (rectangle 10 20 "solid" "blue"))
(define I1 (rectangle 20 20 "solid" "blue"))
(define I2 (rectangle 30 15 "solid" "blue"))

;; Natural is one of:
;; - 0
;; - (add1 Natural)
;; interp. a natural number
(define N0 0)
(define N1 1)

;; ListOfImage is one of:
;; - empty
;; - (cons Image ListOfImage)
;; interp. a list of images
(define LOI0 empty)
(define LOI1 (list I0 I1 I2))
```

In this question you will be asked to design a function `nth-image` that consumes a `Natural` `n` and a `ListOfImage` `loi` and that produces the `nth` image in the `loi` or `false` if no such image exists. So,

```
(nth-image 0 empty)  ; produces false
(nth-image 0 LOI1)   ; produces I0, the 0th image in the list
(nth-image 1 LOI1)   ; produces I1, the 1st image in the list
(nth-image 4 LOI1)   ; produces false
```

Note that this function operates on two complex data.

a) Draw the table representing the cross-products of the types comments and fill it in:

Answer:

```
\begin{array}{ccc}
0 & n & add1 \\
\hline
\text{empty} & false & false \\
\text{lst} & | & \\
\text{cons} & | & \text{(first lst)} \\
\end{array}
```

\[<\text{get \((n-1)\)th image from rest of list}>\] (*)

(*) natural recursion

[4 points - labelled axes
- [1 point for each of 0, add1, empty, cons]]

[4 points - cells
- [1 point for each cell correctly filled in]]
b) Complete the design of the function `nth-image`:

```
;;; Natural ListOfImage -> Image or false
;;; produces the nth image in loi or false if no such image exists
(check-expect (nth-image 0 empty) false)
(check-expect (nth-image 0 LOI1) I0)
(check-expect (nth-image 2 LOI1) I2)
(check-expect (nth-image 4 LOI1) false)

;(define (nth-image n loi) false) ;stub
```

Answer:

```
(define (nth-image n loi)
  (cond [(empty? loi) false]
        [(zero? n) (first loi)]
        [else
          (nth-image (sub1 n) (rest loi))]))
```

[3 points - three clauses
  - [1 point for each correct question]
  - deduct 1 point if there are 4 or more clauses]
[4 points - correct answers
  - [1 point - false]
  - [1 point - (first loi)]
  - [2 points - natural recursion
    - [1 point for each of
      (sub1 n) and (rest loi)]]
[2 points - correspondence between this function definition and the model (the table) built in part (a) – no part marks]
Q5. Consider the following partial data definitions:

```
(define-struct task (name duration))
;; Task is (make-task String Natural)
;; interp. a task with a name and duration in minutes
(define T1 (make-task "pre-lab #2" 30))
(define T2 (make-task "problem set #2" 150))
(define T3 (make-task "homework" 120))
(define T4 (make-task "laundry" 90))
(define T5 (make-task "clean fridge" 300))

(define-struct project (name lop lot))
;; Project is (make-project String ListOfProject ListOfTask)
;; interp. a project with a name, list of sub-projects and
;; list of tasks
(define P0 (make-project "empty" empty empty))
(define P1 (make-project "cpsc" empty (list T1 T2)))
(define P2 (make-project "math" empty (list T3)))
(define P3 (make-project "ubc" (list P1 P2) empty))
(define P4 (make-project "home" empty (list T4 T5)))
(define P5 (make-project "all" (list P3 P4) empty))

;; ListOfProject is one of:
;; - empty
;; - (cons Project ListOfProject)
;; interp. a list of projects
(define LOP0 empty)
(define LOP1 (list P1 P2))

;; ListOfTask is one of:
;; - empty
;; - (cons Task ListOfTask)
;; interp. a list of tasks
(define LOT0 empty)
(define LOT1 (list T1 T2))
```

a) On the types comments above, draw reference arrows from every type reference to the corresponding type definition. The pointy-end of the arrow must always end at a type name before the word 'is'.

Label each arrow with one of:
- **MR** if it is involved in a mutual-reference cycle
- **SR** if it is a self-reference
- **R** otherwise

[6 points - 1 point for each arrow - arrowhead must be in right place]
[6 points - 1 point for each correct label on an arrow]
[DEDUCT 1 point for each extra arrow]
b) Now consider the following template function, formed by encapsulating the template functions for Task, Project, ListOfTask and ListOfProject.

In the template function there are 6 calls to other template functions (the trampoline is not included). We have numbered each of the calls 1 through 6. Go back to the arrows you drew in part a) and label each of them with the corresponding number 1 -> 6. You do not have to add anything to the template functions below.

```
(define (fn-for-project p)
  (local [(define (fn-for-task t)
                (... (task-name t) (task-duration t)))
    ...]

(define (fn-for-project p)
  (... (project-name p)
       (fn-for-lop (project-lop p))   ;[1]
       (fn-for-lot (project-lot p))))  ;[2]

(define (fn-for-lop lop)
  (cond [(empty? lop) (...)]
       [else (... (fn-for-project (first lop))  ;[3]
                  (fn-for-lop (rest lop)))]))   ;[4]

(define (fn-for-lot lot)
  (cond [(empty? lot) (...)]
       [else (... (fn-for-task (first lot))  ;[5]
                   (fn-for-lot (rest lot)))]))  ;[6]

(fnn-for-project p))
```

[6 points - 1 point for each correct]
c) Now design a function that consumes a project and produces the total time needed to complete all tasks in the project (and each of that project's sub-projects, etc.)

Another copy of the template has been provided for you. Simply edit it neatly by crossing things out that are not needed and by clearly showing what you would put in place of the dots. Be sure to include a signature, purpose statement and check-expects. You MUST use the examples from the data definitions above to formulate your check-expects.

Answer:

```
;; Project -> Natural
;; produces the total time to complete all tasks in a project p
(check-expect (time-project P0) 0)
(check-expect (time-project P5) 690)

(define (time-project p)
  (local [[(define (fn-for-project p)
            (+ (fn-for-lop (project-lop p))
               (fn-for-lot (project-lot p)))]])

    (define (fn-for-lop lop)
      (cond [(empty? lop) 0]
            [else
             (+ (fn-for-project (first lop))
                (fn-for-lop (rest lop)))]))

    (define (fn-for-lot lot)
      (cond [(empty? lot) 0]
            [else
             (+ (task-duration (first lot))
                (fn-for-lot (rest lot)))]))

    (fn-for-project p)))
```

[1 point - correct signature (no part marks)]
[1 point - purpose statement]
[2 points - check-expects (first test must test base case)]
[1 point - fn-for-project in global scope given more descriptive name]
[1 point - delete (project-name p) in fn-for-project]
[1 point - use '+' for combination in fn-for-project]
[1 point - use 0 for base case in fn-for-lop]
[1 point - use '+' for combination in fn-for-lop]
[1 point - use 0 for base case in fn-for-lot]
[1 point - use '+' for combination in fn-for-lot]
[1 point - EITHER:
  - rename fn-for-task to task-duration in fn-for-lot AND completely delete fn-for-task
  OR:
  - delete combination step and (task-name t) in fn-for-task]
d) **Note:** this is the most difficult question on the exam. Make sure you have answered all the other questions before spending time on this one.

The following is an abstract `fold-project` function:

```scheme
;; (String X Y -> W) (W X -> X) (Z Y -> Y) (String Integer -> Z)
;; X Y Project -> W
;; fold for project
(check-expect (fold-project make-project cons cons
                make-task empty empty P5) P5)

(define (fold-project c0 c1 c2 c3 b1 b2 p)
  (local [(define (fn-for-task t) ; Task -> Z
               (c3 (task-name t) (task-duration t)))
         (define (fn-for-project p) ; Project -> W
          (c0 (project-name p)
               (fn-for-lop (project-lop p))
               (fn-for-lot (project-lot p))))
         (define (fn-for-lop lop) ; ListOfProject -> X
          (cond [((empty? lop) b1]
                 [else
                  (c1 (fn-for-project (first lop))
                       (fn-for-lop (rest lop)))])]
         (define (fn-for-lot lot) ; ListOfTask -> Y
          (cond [((empty? lot) b2]
                 [else
                  (c2 (fn-for-task (first lot))
                       (fn-for-lot (rest lot)))]])]
          (fn-for-project p)))

Complete the design of the function `task-names` that consumes a project and produces a list of all the names of the tasks in that project (and each of it’s subprojects, etc.) You must use the abstract `fold-project` function provided above along with one or more anonymous functions – you may not use a local expression.

```scheme
;; Project -> ListOfString
;; produces a list of the names of all the tasks in project p
(check-expect (task-names P0) empty)
(check-expect (task-names P5)
  (list "pre-lab #2" "problem set #2" "homework"
        "laundry" "clean fridge")
(define (task-names p) empty) ; stub
Answer:

\[
(define \text{task-names } p) \\
  (\text{fold-project } (\lambda(a \ b \ c) \ (\text{append } b \ c)) \ \text{append cons } (\lambda(a \ b) \ a) \ \\
  \text{empty empty } p))
\]

[1 point - call to \text{fold-project with correct number of parameters}]  
[2 points - c0: (\lambda(a \ b \ c) \ (\text{append } b \ c))]  
[1 point - c1: \text{append}]  
[1 point - c2: \text{cons}]  
[2 points - c3: (\lambda(a \ b) \ a)]  
[1 point - b1: \text{empty}]  
[1 point - b2: \text{empty}]  
[1 point - p: passed as last parameter]

Alternate answer:

c2: \text{append} – but only if c3 produces a list.  
c3: (\lambda(a \ b) \ (\text{list } a))]