YOUR NAME:

Instructions:

1. Please make sure you are sitting at the seat assigned to you.

2. Please write your name on both the exam booklet and the answer sheet.

3. You MUST enter your answers into the answer sheet. Each problem that requires an answer has been numbered. Place your answer at the corresponding number on the answer sheet. Please use the empty space on the exam booklet to show your work.

4. While there may be a lot of reading to do on a problem, there is relatively little code to write, so please take the time to read each problem carefully.

5. Note that any procedures or code fragments that you write will be judged not only on correct function, but also on clarity and good programming practice.

6. Talking, passing around pencils and erasers etc. are not allowed. Please raise your hand if you have a question and a TA will be with you. No questions during the last 30 minutes.

7. You have 120 minutes. Good luck!

[Part 1. Substitution Model (20 points)]

Each question below should be treated as independent. For each, the given sequence of expressions is evaluated in the order shown. Write the value that will be returned for the last expression in each sequence. If the sequence results in a error, write error and indicate the kind of error. If the final expression returns a procedure, write procedure and indicate the type of the procedure.

Question 1.1:

```
(define (f x n)
  (if (= n 0)
      (list 'x)
      (cons x (f x (- n 1)))))

(f 1 3)
```

;;; (1 1 1 x)

Question 1.2:

```
((lambda (f) (lambda (x y) (f y x))) -)
```

;;; procedure: Number,Number -> Number

Question 1.3:

```
(define f
  (lambda (f)
    (lambda (g) (g f))))

((f 5) (lambda (x) (+ x 1)))
```

;;; 6

Question 1.4:

```
(define (repeated f n)
  (if (= n 0)
      (lambda (x) x)
      (lambda (x) (f ((repeated f (- n 1)) x)))))

((repeated (lambda (x) (expt x 2)) 3) 2)
```

;;; 256

[Part 2. Procedures and Orders of Growth]

The arithmetic mean of numbers x1,x2,...,xn is ((x1 + x2 + ... + xn) / n).

Consider the following code:

```
; type: list<number> -> number
(define (mean-1 lst)
  (if (null? lst) 0
      (let ((result (mean-helper lst)))
        (/ (car result) (cadr result)))))

; type: list<number> -> pair<number, pair<number, null>>
(define (mean-helper lst)
  (if (null? lst)
      (list 0 0)
      (let ((info-from-rest (mean-helper (cdr lst))))
        (list (+ (car lst) (car info-from-rest))
              <EXP A>
              ))))
```

Question 2.1: Write the code that goes in place of <EXP A> so that the procedure mean-1 correctly computes the arithmetic mean of a list of numbers.

```
;;; (+ 1 (cadr info-from-rest))
```

Question 2.2: The method of computing the arithmetic mean used by the code on this page is:

* Choice A: an iterative algorithm
* Choice B: a recursive algorithm
* Choice C: cannot determine from the information given

;;; B

An alternate way of computing the mean is to use the following procedures to compute the list sum and length separately:

```
(define (sum lst) ; type: list<number> -> number
  (define (help lst sofar)
    (if (null? lst) sofar
        (help (cdr lst) (+ (car lst) sofar))))
  (help lst 0))

(define (length lst) ; type: list<anytype> -> number
  (define (help lst sofar)
    (if (null? lst) 0
        (+ 1 (help (cdr lst) sofar))))
  (help lst 0))

(define (mean-2 lst)
  (if (null? lst) 0
      (/ (sum lst) (length lst))))
```

If n is the length of the list, we are interested in the order of growth of the processes generated by these procedures.

* Choice A: a linear iterative process, i.e. requires an order of growth in time (the number of addition operations) that is linear in n for large n, and an order of growth in storage space (the stack up of deferred operations) that is constant for large n.

* Choice B: a linear recursive process, i.e. requires an order of growth in time that is linear in n for large n, and an order of growth in storage space that is also linear in n for large n.

* Choice C: a recursive process that requires order of growth greater than linear in time or more than linear in space for large n.

* Choice D: an iterative process that requires greater than linear time order of growth in n but constant space for large n.

* Choice E: a constant space and time process which requires computation that does not grow in proportion to n.

* Choice F: cannot determine from the information given.

Question 2.3: What is the order of growth for the process resulting from the sum procedure?
;;; Choice: A

Question 2.4: What is the order of growth for the process resulting from the length procedure?
;;; Choice: B

Question 2.5: What is the order of growth for the process resulting from the mean-2 procedure?
;;; Choice: B

[Part 3. Pairs and Lists (20 points)]

The following is intended to count the number of pairs in a list structure.

```
(define (count-pairs x)
  (if (not (pair? x))
      0
      (+ (count-pairs (car x))
         (count-pairs (cdr x))
         1)))
```

This procedure will NOT correctly count the number of pairs. Draw box-and-pointer diagrams of list structures with exactly three pairs, for which this procedure would behave as follows.

Question 3.1: Procedure would return 3.
;;; A list of 3 pairs.

Question 3.2: Procedure would return 4.
;;; A list of three pairs, where the car of the second pair points to the third.

Question 3.3: Procedure would return 7.
;;; A list of three pairs, where the car of the first points to the second, and
;;; the car of the second points to the third.

Question 3.4: Procedure would never return.
;;; A list of three pairs, where the car of the last points to the first.

[Part 4. Abstract Data Types (20 points)]

We are going to explore a new kind of data structure, called a cycle. You can think of this as a kind of circular list: indeed we are going to represent a cycle as a loop of cons cells, where the car of each cell points to a value, and the cdr of each cell points to the next element in the circular sequence. To create a cycle from a list, we can use:

```
(define (list->cycle lst)
  (set-cdr! (last lst) lst)
  lst)

(define (last lst)
  (if (null? lst)
      (error "not long enough")
      (if (null? (cdr lst))
          lst
          (last (cdr lst)))))
```

For example:
```
(define test-cycle (list->cycle '(a b c g)))
```

To see what this structure looks like, you might find it convenient to draw a box-and-pointer diagram in the space below.

Associated with a cycle, we have several operations:

* head will return the value stored at the cell in the cycle pointed to by its argument, e.g. (head test-cycle) will return the value a;

* rotate-left will rotate the cycle one element to the left, e.g., (head (rotate-left test-cycle)) will return the value b; (you can think of a cycle as a circular list in which the elements are arranged in clockwise sequence, the head is at the top, and "left" means rotating the sequence counter-clockwise);

* rotate-right will rotate the cycle one element to the right, in analogy to rotate-left, e.g., (head (rotate-right test-cycle)) will return the value g.

Defining head is easy:

```
(define head car)
```

Question 4.1: Write the procedure rotate-left. Be sure that it returns a pointer to the correct cell in the sequence. You may assume that the argument is a non-empty cycle.

```
;;; (define (rotate-left cycle) (cdr cycle))
```

Question 4.2: Write the procedure rotate-right. Be sure that it returns a pointer to the correct cell in the sequence. You may assume that the argument is a non-empty cycle.

```
;;; (define (rotate-right cycle)
;;;   (define (aux where start)
;;;     (if (eq? (cdr where) start)
;;;         where
;;;         (aux (cdr where) start)))
;;;   (aux cycle cycle))
```

Now we want to add a new element to a cycle. This means we want to insert a new cons cell, with the supplied argument as its car, before the location in the cycle currently pointed to by the supplied argument. We are going to treat this as a mutator (that is, it changes an existing data structure), so we do not want to rely on the value returned by the procedure. Hence, by convention, let's have the procedure return the symbol 'done. Here is an example behavior:

```
(define test-cycle (list->cycle '(a b c g)))
;Value: test-cycle

(head test-cycle)
;Value: a

(head (rotate-left (rotate-left test-cycle)))
;Value: c

(insert-cycle! (rotate-left (rotate-left test-cycle)) 'x)
;Value: done

(head test-cycle)
;Value: a

(head (rotate-left (rotate-left test-cycle)))
;Value: x

(head (rotate-left (rotate-left (rotate-left test-cycle))))
;Value: c
```

Question 4.3: Write the procedure insert-cycle. You may assume that the argument is a non-empty cycle. You should use rotate-left and/or rotate-right where appropriate.

```
;;; (define (insert-cycle! new cycle)
;;;   (let ((new-cell (list new)))
;;;     (set-cdr! new-cell cycle)
;;;     (set-cdr! (rotate-right cycle) new-cell)
;;;     'done))
```

Finally, suppose we want to remove an element from the cycle. Our goal is to remove from the cycle the cell to which the supplied argument points, as well as any pointers into the cycle from removed cells. Note that if we remove the cell to which a global variable refers, we may break our data structure, but we will assume that it is the responsibility of the user to manage pointers into the cycle.

Question 4.4: Provide a definition for the procedure delete-cycle!. Use rotate-right, rotate-left, head and set-cdr! but no other list operations. Also be sure that you change the cdr of the cycle cell that contained the deleted value to the empty list.

```
;;; (define (delete-cycle! cycle)
;;;   (set-cdr! (rotate-right cycle) (rotate-left cycle))
;;;   (set-cdr! cycle '())
;;;   'done)
```

[Part 5. Environment Model (20 points)]

Consider the environment diagram that results from evaluating the following expressions in this order. We suggest you draw your own environment diagram in the space below, in order to answer the following questions. Indicate the global environment by GE, and then be sure to label any additional environments, E1, E2, etc, in the order in which they are created.

```
(define (snoc x y) (lambda (m) (m x y)))
(define (rac z) (z (lambda (p q) p)))
(define x (snoc (quote a) (quote b)))
(define result (rac x))
```

Find the following procedure objects in your diagram and give them these labels (i.e. p1, p2, p3 or p4):

* P1 is the double bubble with parameters p, q and body p.
* P2 is the double bubble with parameters x, y and body (lambda (m) (m x y)).
* P3 is the double bubble with parameters m and body (m x y).
* P4 is the double bubble with parameters z and body (z (lambda (p q) p)).

For each of the following questions, answer with a symbol, or a procedure object label (P1, P2, P3, P4) corresponding to your environment diagram.

The global environment has bindings for snoc, rac, x and result.
Question 5.1: What is the value bound to snoc?
;;; p2
Question 5.2: What is the value bound to rac?
;;; p4
Question 5.3: What is the value bound to x?
;;; p3
Question 5.4: What is the value bound to result?
;;; a

The first frame of environment E1 has bindings for x and y.
Question 5.5: What is the value bound to x?
;;; a
Question 5.6: What is the value bound to y?
;;; b

The first frame of environment E2 has a binding for z.
Question 5.7: What is the value bound to z?
;;; p3

The first frame of environment E3 has a binding for m.
Question 5.8: What is the value bound to m?
;;; p1

The first frame of environment E4 has bindings for p and q.
Question 5.9: What is the value bound to p?
;;; a
Question 5.10: What is the value bound to q?
;;; b

Indicate the enclosing environment for each procedure argument:
Question 5.11: p1:
;;; E2
Question 5.12: p2:
;;; GE
Question 5.13: p3:
;;; E1
Question 5.14: p4:
;;; GE

Indicate the enclosing environment for each environment:
Question 5.15: E1:
;;; GE
Question 5.16: E2:
;;; GE
Question 5.17: E3:
;;; E1
Question 5.18: E4:
;;; E2