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. Environment model (10 points)]

According to the environment model:

Question 1.1 (1 pt): What does define do if the variable defined is not bound in the current environment?

;;; Creates a new binding in the first frame.

Question 1.2 (1 pt): What does define do if the variable defined is bound in the first frame of the current environment?

;;; Overwrites the binding in the first frame.

Question 1.3 (1 pt): What does define do if the variable defined is bound in the second frame of the current environment?

;;; Creates a new binding in the first frame, shadowing the second frame.

Question 1.4 (1 pt): What does set! do if the variable modified is not bound in the current environment?

;;; Gives an unbound variable error.

Question 1.5 (1 pt): What does set! do if the variable modified is bound in the first frame of the current environment?

;;; Overwrites the binding in the first frame.

Question 1.6 (1 pt): What does set! do if the variable modified is bound in the second frame of the current environment?

;;; Overwrites the binding in the second frame.

Question 1.7 (2 pts): According to the lambda rule, where does the environment pointer of a newly created procedure point to?

;;; The first frame of the environment in which the procedure is created.

Question 1.8 (2 pts): According to the application rule, where does the enclosing environment pointer of a newly created frame point to?

;;; The target of the environment pointer of the procedure for which
;;; the new frame was created.

[Part 2. Evaluation (10 points)]

Each of the following questions 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 an 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 2.1:

```
(define (test + a b)
  (a + b))

(test 4 − 3)

;;; 1
```

Question 2.2:

```
(define x 5)

(let ((x 3)
      (y (* x 2)))
  (list x y))

;;; (3 10)
```

Question 2.3:

```
(define a 1)
(define b 2)
(define c 3)

(list (list a 'b c)
      '(a b c))

;;; ((1 b 3) (a b c))
```

Question 2.4:

```
(lambda (x)
  (lambda (y)
    (+ x y)))

;;; Procedure: Number −> (Number −> Number)
```

Question 2.5:

```
(define a (list 1 2))
(define b (list 'a 'b))
(define c (list a b))

(set−cdr! (cadr c) nil)
(set−cdr! (cdar c) (cadr c))

c

;;; ((1 2 a) (a))
```

[Part 3. Environment diagrams (28 points)]

Consider the following sequence of expressions:

```
(define (monitor proc)
  (let ((args '())
        (calls 0))
    (lambda (n)
      (set! args (cons n args))
      (set! calls (+ 1 calls))
      (cond ((number? n) (proc n))
            ((eq? n 'calls) calls)
            ((eq? n 'args) args)))))

(define m-sq (monitor (lambda (x) (* x x))))

(m-sq 4)

(m-sq 6)
```

Suppose we trace out with an environment model the evaluation of these expressions. Attached at the end of the quiz is a partially completed environment diagram. Note that each procedure and each frame is labeled, however these labels do not reflect the order in which these elements were generated. You are to use these labels to answer the questions below, specifically you should complete this diagram, and then use the labels on the parts of the environment diagram to answer the following questions.

Question 3.1 (10 pts): For each of the following frames, indicate the lowest frame of the enclosing environment: GE, E1, E2, E3, E4. Each answer should indicate a frame label or "none" or "not shown".

;;; GE->none, E1->E2, E2->E4, E3->E2, E4->GE

Question 3.2 (8 pts): For each of the following procedure objects, indicate the lowest frame of the enclosing environment, choosing one of GE, E1, E2, E3, E4 or none or not shown:

P1, P2, P3, P4.

;;; P1->GE, P2->E2, P3->GE, P4->E4

Question 3.3 (10 pts): For each of the following variable names, indicate the value to which it is bound in the specified environment at the end of the evaluation of the expressions. Indicate the value by choosing one of GE, E1, E2, E3, E4, E5 or P1, P2, P3 or a symbol, a number, a list of numbers or a boolean value:

monitor	GE
m-sq	GE
proc	E4
args	E2
calls	E2

```
;;; monitor | GE = P3
;;; m-sq    | GE = P2
;;; proc    | E4 = P1
;;; args    | E2 = (6 4)
;;; calls   | E2 = 2
```

[Part 4. Doubly linked lists (28 pts)]

We have used lists as a very convenient data structure many times during the term. Our traditional lists are based on using cons cells to glue things together. In this part, we are going to create a doubly-linked list. This means that each element in the list points to both the next element in the list and the previous element. Thus, we can easily move in either direction along the list.

To do this, we are going to implement such a list using objects with state. Here is the procedure to create an element of such a double-linked list:

```
(define (create-node value)
  (let ((right #f)
        (left #f))
    (lambda (msg)
      (cond ((eq? msg 'value) value)
            ((eq? msg 'right) right)
            ((eq? msg 'left) left)
            ((eq? msg 'set-right!)
             (lambda (new)
               (set! right new)))
            ((eq? msg 'set-left!)
             (lambda (new)
               (set! left new)))
            (else (error "don't know message" msg))))))
```

Thus, for example:

```
>(define test (create-node 1))

>(test 'value)
1

>(test 'right)
#f
```

We now want to take a traditional list as input, and create a double-linked list, composed of these message-passing nodes.

```
(define (create-double-list lst)
  (let ((set-of-nodes (map create-node lst)))
    (connect set-of-nodes 'right)
    (connect set-of-nodes 'left)
    (car set-of-nodes)))
```

To do this, we need to write the procedure connect.

```
(define (connect set dir)
  (cond ((null? (cdr set)) 'done)
        ((eq? dir 'right)
         ANSWER4.1)
        ((eq? dir 'left)
         ANSWER4.2)
        (else (error "unknown direction"))))
```

Question 4.1 (5 pts): What expression(s) should be used for ANSWER4.1? Describe your answer briefly in English and provide your code.

```
;;; Answer: send message to rst element to set right pointer to next element,
;;; then continue to connect down cdr of list

;;; (((car set) 'set-right!) (cadr set))
;;; (connect (cdr set) dir)
```

Question 4.2 (5 pts): What expression(s) should be used for ANSWER4.2? Describe your answer briefly in English and provide your code.

```
;;; Answer: same idea as above

;;; (((cadr set) 'set-left!) (car set))
;;; (connect (cdr set) dir)
```

Suppose we have evaluated the following:

```
(define dlist (create-double-list '(1 2 3 4 5)))
```

and we want to use such a list in a manner similar to normal lists, i.e., we can look at the dcar (or the "car" of a double-list) or the dcdr (or the "cdr" of a double-list), or we could look at the dcur which would be the same as a dcdr but moving in the opposition direction, i.e. toward the front of the double-linked list.

```
>(dcar dlist)
1

>(dcar (dcdr dlist))
2

>(dcadr dlist)
2

>(dcar (dcur (dcdr dlist)))
1
```

Question 4.3 (2 pts): Write the definition for dcar.

```
;;; (define dcar (lambda (dlist) (dlist 'value)))
```

Question 4.4 (2 pts): Write the definition for dcdr.

```
;;; (define dcdr (lambda (dlist) (dlist 'right)))
```

Question 4.5 (2 pts): Write the definition for dcur.

```
;;; (define dcur (lambda (dlist) (dlist 'left)))
```

Question 4.6 (2 pts): Write the definition for dcadr.

```
;;; (define dcadr (lambda (dlist) ((dlist 'right) 'value)))
```

Given the ability to traverse a list in either direction, write a mapdouble function that works like map, but on doubly-linked lists, and takes three arguments: (mapdouble fn lst dir), where the last argument indicates the direction in which to traverse lst, applying fn to each element of lst and producing a doubly-linked list of the results. Note that if the initial value for lst points to an element midway along a double list, the result of mapdouble should include only instances of elements from that point to the end of the list in the specified direction, and should point to the corresponding list element in the new list. For example, if we do

```
(define db1 (create-double-list '(1 2 3 4))
```

then

```
(mapdouble square (dcdr (dcdr db1)) 'left)
```

should return a node whose value is 9, whose left node has the value 4, etc. (rather than returning the same doubly-linked list by pointing to the node whose value is 1). You may find the following procedures useful, and you may also find it useful to use reverse which reverses a normal list.

```
(define (get-last lst dir)
  (if (not (lst dir))
      lst
      (get-last (lst dir) dir)))

(define (get-values lst dir)
  (if (not lst)
      '()
      (cons (dcar lst)
            (get-values (lst dir) dir))))
```

Question 4.7 (10 pts): Write the procedure mapdouble

```
;;; (define (mapdouble fun lst dir)
;;;   (if (eq? dir 'right)
;;;       (create-double-list (map fun (get-values lst dir)))
;;;       (get-last
;;;        (create-double-list (reverse (map fun (get-values lst dir))))
;;;        'right)))
```

[Part 5. Abstract Data Types (24 pts)]

In this question, we are going to be exploring a simple system for keeping tracking of voting results, for use in the upcoming election. The basic element of the system will be a record of votes for a candidate in a specific precinct (or voting area).

While the questions all follow a common theme and build on one another, they are all independent: Even if you get a question wrong, or leave it blank, you can still go on to the rest of the questions and answer them. In those later questions you can use the names of the functions you were asked to write in earlier questions, whether or not you got that previous question correct.

First we need to worry about a data structure for representing this information. We will assume that we have a constructor called precinct−data which takes as arguments a name of a candidate (represented as a string), a number of votes for that candidate, and the number of the precinct, in that order. For example

```
(define data1 (precinct−data "fred" 203 1))
(define data2 (precinct−data "judy" 253 1))
(define data3 (precinct−data "fred" 102 2))
(define data4 (precinct−data "judy" 203 2))
(define data5 (precinct−data "fred" 193 3))
(define data6 (precinct−data "judy" 143 3))

(define test−votes (list data1 data2))  ; simple list case

(define allvotes                        ; larger test case
  (list data1 data2 data3 data4 data5 data6))
```

Associated with this constructor are several selectors or accessors: who, votes and precinct each extract the associated parts of a precinct count.

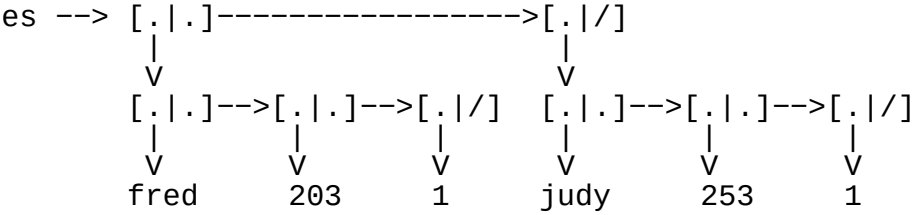
Question 5.1 (8 pts): Write an implementation for the constructor and these accessors.

```
;;; (define (precinct−data who votes precinct)
;;;   (list who votes precinct))

;;; (define who car)
;;; (define votes cadr)
;;; (define precinct caddr)
```

Question 5.2 (8 pts): In the answer sheet, draw a box−and−pointer diagram for your implementation for the structure bound to test−votes.

```
(define data1 (precinct−data "fred" 203 1))
(define data2 (precinct−data "judy" 253 1))
(define test−votes (list data1 data2)) ; simple list case
```



Assume that the procedure addup exists, with the following behavior: it takes as argument a list of numbers, and it returns as value the sum of those numbers. With that assumption, we want to write a procedure, called get−votes−for, which takes as arguments a name of a candidate (as a string) and a list of precinct data objects (such as the example shown for allvotes). It returns as value the sum of the votes from all precincts for the candidate. We have provided a template below:

```
(define (get−votes−for cand data)
  (addup
   (map ANSWER5.3
        (filter ANSWER5.4 data))))

where

(define (map proc lst)
  (if (null? lst)
      '()
      (cons (proc (car lst))
            (map proc (cdr lst)))))

(define (filter pred lst)
  (cond ((null? lst) '())
        ((pred (car lst)) (cons (car lst) (filter pred (cdr lst))))
        (else (filter pred (cdr lst)))))
```

Question 5.3 (4 pts): Complete the code needed for ANSWER5.3.

Question 5.4 (4 pts): Complete the code needed for ANSWER5.4. (you may find it useful to use string=?)

```
;;; Here is the completed code for both parts
;;; (define (get−votes−for cand data)
;;;   (addup
;;;    (map VOTES
;;;     (filter (LAMBDA (X) (STRING=? (WHO X) CAND)) data))))
```