CS 61A, Fall, 2005, Midterm 3, Harvey

This exam is worth 40 points, or about 13% of your total course grade. It includes two parts: The individual exam (this part) is worth 34 points, and the group exam (the other part you probably just finished – not included in this PDF) is worth 6 points. The individual part contains six substantive questions.

**When writing procedures, don't put in error checks. Assume that you will be given arguments of the correct type.**

Our expectation is that many of you will not complete one or two of these questions. If you find one question especially difficult, leave it for later; start with the ones you find easier.

**Question 1 (6 points):**

What will the Scheme interpreter print in response to **the last expression** in each of the following sequences of expressions? Also, draw a "box and pointer" diagram for the result of each printed expression. If any expression results in an error, **circle the expression that gives the error message.** Hint: It'll be a lot easier if you draw the box and pointer diagram *first*!

```
(define f (list 2 3))
(define g (append f f))
(set-car! g f)
g
```

```
(let ((x (list 1 2 3)))
     (set-car! x (list 'a 'b 'c))
     (set-car! (cdar x) 'd)
     x)
```

```
(define x 3)
(define m (list x 4 5))
(set! x 6)
m
```

**Question 2 (6 points):**

(a) Here are some situations that might be simulated using OOP. In each case we want to know whether class A should be a parent of class B (answer "Yes" or "No"):

- We're simulating a kitchen. Class A: silverware. Class B: fork.

- We're simulating a shopping mall. Class A: food court. Class B: restaurant.

- We're simulating a library. Class A: bookshelf. Class B: book.

(b) In each of the following situations, should the given variable be a class variable or an instance variable (answer "Class" or "Instance")?

- In the shoe class, the total number of shoes in the world.

- In the refrigerator class, the maximum safe temperature.

- In the person class in the adventure game, the person's favorite color.

**Question 3 (3 points):**

What are the possible values of variable `a` after each of the following parallel executions? **If a deadlock is a possibility, say so.**

(a)

```
(define a 2)
(parallel-execute
     (lambda () (set! a (list a a)))
     (lambda () (set! (list a 1))))
```

(b)

```
(define a 2)
(define s (make-serializer))
(parallel-execute
     (s (lambda () (set! a (list a a))))
     (s (lambda () (set! a (list a 1)))))
```

(c)

```
(define a 2)
(define s (make-serializer))
(define t (make-serializer))
(parallel-execute
     (s (lambda () (set! a (list a a))))
     (t (lambda () (set! a (list a 1)))))
```

**Question 4 (7 points):**

Suppose there are N students taking a midterm. Suppose we have a vector of size N, and each element of the vector represents one student's score on the midterm. Write a procedure `(histogram scores)` that takes this vector of midterm scores and computes a histogram vector. That is, the resulting vector should be of size M+1, where M is the maximum score on the midterm (it's M+1 because scores of zero are possible), and element number I on the resulting vector is the number of students who got score I on the midterm.

For example:

```
> (histogram (vector 3 2 2 3 2))
#(0 0 3 2) ;; no students got 0 points, no students got 1 point,
           ;; 3 students got 2 points, and 2 students got 3
           ;;points.
> (histogram (vector 0 1 0 2))
#(2 1 1)  ;; 2 students got 0 points, 1 student got 1 point,
          ;; and 1 student got 2 points.
```

**Do not use** `list->vector` **or** `vector->list`.

Note: You may assume that you have a procedure `vector-max` that takes a vector of numbers as an argument, and returns the largest number in the vector.

**Question 5 (4 points):**

**On the next page is the relevant code from the Scheme-2 interpreter.**

Louis Reasoner wants to make the following change to `eval-2`:

```
(define (eval-2 exp env)
     (cond ...
          ((define-exp? exp)
           (put (cadr exp) (eval-2 (caddr exp) THE-GLOBAL-ENV)
               env)
          'okay)
          ...))
```

(a) In one English sentence, describe precisely the programs that work in the real Scheme-2, but don't work in Louis's version.

(b) Give a short example of a procedure that works in the real Scheme-2, but doesn't work in Louis's version.

**Reference code for question 5:**

Here is the *real* Scheme-2 code, *not* Louis's version.

```
(define (eval-2 exp env)
    (cond ((constant? exp) exp)
          ((symbol? exp) (lookup exp env))
          ((quote-exp? exp) (cadr exp))
          ((if-exp? exp)
           (if (eval-2 (cadr exp) env)
               (eval-2 (caddr exp) env)
               (eval-2 (cadddr exp) env)))
          ((lambda-exp? exp) (make-proc exp env))
          ((define-exp? exp)                    ;Louis's change
           (put (cadr exp) (eval-2 (caddr exp) env) ;goes here.
                env)
           'okay)
          ((pair? exp) (apply-2 (eval-2 (car exp) env)
                                (map (lambda (e) (eval-2 e env))
                                     (cdr exp))))
          (else (error "bad expr: " exp))))

(define (apply-2 proc args)
     (cond ((procedure? proc)
            (apply proc args))
           ((lambda-proc? proc)
            (eval-sequence (proc-body proc)
                           (extend-environment (proc-params proc)
                                               args
                                               (proc-env proc))))
     (else (error "bad proc: " proc))))
```

**Question 6 (7 points):** For reference, here is the insertion sort program you saw in week 3, rewritten to use lists instead of sentences:

```
(define (sort lst)
     (if (null? lst)
     '()
     (insert (car lst) (sort (cdr lst)))))

(define (insert value sorted)
     (cond ((null? sorted) (list value))
          ((< value (car sorted)) (cons value sorted))
          (else (cons (car sorted) (insert value (cdr sorted))))))
```

We are going to rewrite this to use list mutation, rearranging the order of the pairs in the spine of the argument list; **the procedure will return the sorted list.** Don't worry about preserving the value of any variables that point to the original list. Sample use:

```
> (sort! (list 7 3 87 5))
(3 5 7 87)
```

Your job is to fill in the blanks in the partial definition below. You probably will not need all the space given. **Note that the first argument to `insert!` is the pair whose car is the value to be inserted, not the value itself.**

**Your procedure must not create new pairs!**

```
(define (sort! lst)
     (if (null? lst)
         '()
         (insert! lst (sort! (cdr lst)))))

(define (insert! value-pair sorted)
     (cond ((null? sorted)
            (set-cdr! value-pair '())
            value-pair)
           ((< (car value-pair) (car sorted))
            _____________________________________________
            _____________________________________________
            _____________________________________________)
            (else
            _____________________________________________
            _____________________________________________
            _____________________________________________)))
```