CS 61a

Spring 2005

Question 1 (6 points)

What will the scheme interpreter print in response to **the last expression** in each of the following sequence of expressions? Also, draw a "box and pointer" diagram for the result of each sequence. 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 y (list 1 2))
(define z (list 3 4))
(set-car! y z)
(set-cdr! (car y) (cddr y))
y
```

```
(define f (cons 1 2))
(define g (list f f ))
(set! f 3)
(set-car! (car g) 4)
g
```

```
(define h (list 1 2))
(set-cdr! (cdr h) h)
(set-cdr! (cddddr h) 3)
h
```

Question 2 (7 points)

Write a procedure **link-first!** that takes a non-empty list as its argument and destructively links together all elements in the list with the same value as the car of the list. For example,

```
>(define ls '( 1 2 3 2 1 1 2 1))
>(link-first! ls)
>ls
(1 1 1 1)
```

you can assume list elements are numbers as in this example (so don't worry about sublists), but don't use word/sentence procedures.

Your procedure must not create new pairs!

Write your procedure by filling in the blank spaces in the following partly-written definition. Don't define other procedures. We have a particular implementation in mind- note the context in which **helper** is called at the bottom. you will not need as many lines of code as we've provided room for.

```
(define (lin-first! L)
  (define (helper first ls)
     (cond ((null? ls) '())
           (equal? (car ls) first)

                                                    )
           (else

                                                    )))
  (set cdr! L (helper(car L) (cdr L)))
  L)
```

Question 3 (8 points)

Because of the risk of terrorist activity in the Adventure world, the government wishes to know where everyone is at all times. To this end, they maintain a global variable **people** whose values is an association list in which each key is a person and each value s the place where that person is.

People are added to this list as they're created:

```
(define-class (person name place)
  …
 (initialize
  (SET! PEOPLE (CONS (CONS SELF PLACE) PEOPLE))
  (ask self 'put strength 100)
  (ask place 'enter self))
 …)
```

To keep this list up to date, the goventment sends a copy to each place. A spy is a kind of person who notices when another person enters his place, and updates the person's entry in the **people** list.

(a) implement the spy class. For your convenience, the relevant code from the project (including solutions is on the last page of the exam. Don't create new pairs when updating the **people** list.

(question 3 continues)

(b) write a procedure **whereis** that takes a person's name as its argument and returns the name of the place where that person is, according to the **people** list. Assume that there is exactly one person with the given name in the **people** list.

Question 4 (6 points)

Write a procedure **subvector** that takes three arguments: a vector **vec** and two nonnegative integers **start** and **end** that are less than the length of **vec**. It should return a new vector containing only the elements of **vec** at positions between start and end, inclusive:

>(subvector '#(we all live in a yellow submarine) 2 5)

#(live in a yellow)

if end is less than start, return an empty vector.

DO NOT USE list->vector or vector->list.

Question 5 (6 points)

Suppose we do this

```
>(define a 10)
>(define (change-a! proc)
   (set! a (proc a)))
```

(a) What are all the possible values of **a** after the following call to parallel-execute?

```
>(parallel-execute (lambda () (change-a! (lambda (x) ( + x 1))))
                   (lambda () (change-a! (lambda (x) (* x 2)))))
```

(b) Of those, which is/are the value(s) that a correctly serialized version could produce?

The following are attempts to serialize the operations. Dose each properly serialize the operations?

(c)

```
>(define s (make-serializer))
>(parallel-execute (s (lambda () (change-a! (lambda (x) ( + x 1)))))
                   (s (lambda () (change-a! (lambda (x) (* x 2))))))
```

Yes, correct No, not correctly serialized

(d)

```
>(define s (make-serializer))
>(parallel-execute (lambda () (change-a! (s (lambda (x) ( + x 1)))))
                   (lambda () (change-a! (s (lambda (x) (* x 2))))))
```

Yes, correct No, not correctly serialized

(e)

```
>(define s (make-serializer))
>(parallel-execute (lambda () ((s change-a!) (lambda (x) ( + x 1))))
                   (lambda () ((s change-a!) (lambda (x) (* x 2)))))
```

Yes, correct No, not correctly serialized

This excerpt from the Adventure game has irrelevant methods left out.

```
(define-class (place name)
  (parent (basic-object))
  (instance-vars
   (directions-and-neighbors '())
   (things '())
   (people '())
   (entry-procs '())
   (exit-procs '()))
  (method (neighbors) (map cdr directions-and-neighbors))
  (method (exits) (map car directions-and-neighbors))
  (method (enter new-person)
    (if (memq new-person people)
        (error "Person already in this place" (list name new-person)))
    (set! people (cons new-person people))
    (for-each (lambda (proc) (proc)) entry-procs)
    'appeared)
  (method (exit person)
    (for-each (lambda (proc) (proc)) exit-procs)
    (if (not (memq person people))
        (error "Disappearing person not here" (list name person)))
    (set! people (delete person people))
    'disappeared))
```

```
(define-class (person name place)
  (instance-vars
   (possessions '())
   (saying "")
   (money 100))
  (initialize
   (ask self 'put 'strength 100)
   (ask place 'enter self))
  (method (talk) (print saying))
  (method (set-talk string) (set! saying string))
  (method (exits) (ask place 'exits))
  (method (notice person) (ask self 'talk))
  (method (go direction)
    (let ((new-place (ask place 'look-in direction)))
      (cond ((null? new-place)
             (error "Can't go" direction))
            (else
             (ask place 'exit self)
             (announce-move name place new-place)
             (for-each
              (lambda (p)
                (ask place 'gone p)
                (ask new-place 'appear p))
              possessions)
             (set! place new-place)
             (ask new-place 'enter self))))) )
```