CS 3 PRACTICE Final Exam Summer 2008

Read and fill in this page now

Solutions!!!

Name:	
Instructional login (eg, cs3-ab)	
UCWISE login:	
Lab section (day and time):	
T.A.:	
Name of the person sitting to your **left**:	
Name of the person sitting to your **right**:	

	Prob	Score
(3)	Prob 1a	
(2)	1b	
(2)	1c	
(5)	2	
(5)	3	
(5)	4	
Raw Total (out of 22)		
Scaled Total		

You have 3 hours to finish this test, but there should only be about 2 hours worth of material. Your exam should contain 6 problems (numbered 0-5) and 1 appendix on 7 total pages.

This is an open-book test. You may consult any books, notes, or other paper-based inanimate objects available to you. Read the problems carefully. If you find it hard to understand a problem, ask us to explain it.

Restrict yourself to Scheme constructs covered in this course (i.e., in the lab materials, case studies, book chapters that were assigned, and lectures). Use descriptive names in all your code. Remember, use sentence operators only on sentences, not lists!

Please write your answers in the spaces provided in the test; if you need to use the back of a page make sure to clearly tell us so on the front of the page.

Partial credit will be awarded where we can, so do try to answer each question.

Good Luck!

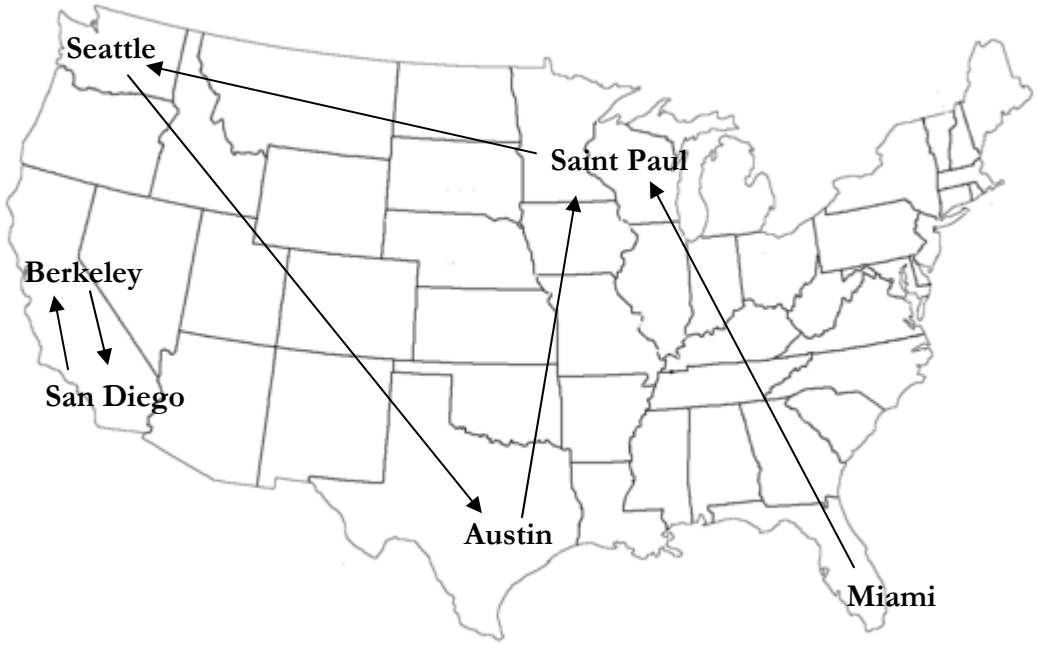
Problem 0: Your name, please! (1 point)

Put your instructional login name (e.g., cs3-ab) on the top of each page.

Problem 1: Loop-de-Loop (Part A: 3 points, Part B: 2 points, Part C: 2 points)

Gilbert has opened up a side business while he is in school... He's running a new hip airline where every seat comes with a computer (rather than a TV). He has a bunch of flights lined but he needs to figure out if once he flies someone somewhere they can get home... He has an association list of all of his flights. There is only ever ONE flight out of an airport...

```
(define *flights*
          '(( Berkeley (San Diego))
            ((San Diego) Berkeley)
            (Seattle Austin)
            (Austin (Saint Paul))
            (Miami (Saint Paul))
            ((Saint Paul) Seattle)))
```



Someone has written a buggy version of `can-get-home.`

```
(define (can-get-home? home flights current-location)
  (let ((next-location (assoc current-location flights)))
    (cond
     ((not next-location) #f)
     ((equal? (cadr next-location) home) #t)
     (else
      (can-get-home home flights (cadr next-location))))))
```

(continued on next page)

```
(define (can-get-home? home flights current-location)
  (let ((next-location (assoc current-location flights)))
    (cond
     ((not next-location) #f)
     ((equal? (cadr next-location) home) #t)
     (else
      (can-get-home home flights (cadr next-location))))))
```

Predict the output

`(can-get-home? 'Berkeley *flights* 'Berkeley)` ➔ **#t**

`(can-get-home? 'Seattle *flights* 'Seattle)` ➔ **#t**

`(can-get-home? 'Miami *flights* 'Miami)` ➔ **error**

Part B: Describe the bug(s)

This procedure works unless there is a loop that does not contain the original node. In the case of Miami this will continually recurse through

`(can-get-home? 'Miami *flights* '(Saint Paul))`

`(can-get-home? 'Miami *flights* 'Seattle)`

`(can-get-home? 'Miami *flights* 'Austin)`

Part C: Describe how you might fix the bug(s)

We could keep track of all the places we've visited through accumulating recursion. Before we decide to continue using the recursive case, we see if we have already visited this location. If we detect a loop in this way, we should return false.

Problem 2: (5 points)
Write a procedure `airplane-routes` that returns a list representing the path from a particular location. The last element in the list should be the original location, unless the original location can not be reached. If a loop occurs, it should contain each of the cities in the loop only once.

```
(airplane-routes 'Berkeley *flights*) ➔
                                        '((San Diego) Berkeley)

(airplane-routes 'Seattle *flights*) ➔
                                      '(Austin (Saint Paul) Seattle)

(airplane-routes 'Miami *flights*) ➔
                                      '((Saint Paul) Seattle Austin)
```

```
(define (airplane-routes home flights)
     (routes-helper home home flights '()))

(define (routes-helper home current-loc flights route-so-far)
  (let ((next-location (assoc current-loc flights)))
    (cond
     ((not next-location) route-so-far)
     ((equal? (cadr next-location) home)
          (append route-so-far (cdr next-location)))
     ((member next-location route-so-far) route-so-far)
     (else (routes-helper
         home
         (cadr next-location)
         flights
         (append route-so-far (list (cadr next-location)))))))
```

Problem 3: (5 points)

Write a procedure `parenthesis-to-word-ratio` that returns the total number of words in a list divided by the total number of parentheses in a list.

```
(parenthesis-to-word-ratio '(()))➔ 0

(parenthesis-to-word-ratio '((a)))➔ 1/4 or .25

(parenthesis-to-word-ratio '( a ( ( x ) a ) (b c d e f) )➔ 1
```

```
(define (parentheses-to-word-ratio L)
  (/ (count-words L) (count-parens L)))


(define (count-words L)
  (cond
   ((null? L) 0)
   ((list? (car L)) (+ (count-words (car L)) (count-words (cdr L))))
   ((word? (car L)) (+ 1 (count-words (cdr L))))))


(define (count-parens L)
  (cond
   ((null? L) 2)
   ((list? (car L)) (+ (count-parens (car L)) (count-parens (cdr L))))
   ((word? (car L)) (count-parens (cdr L)))))
```

Problem 4: Bowling with Lists (5 points)

Write the procedure `bowling-helper` that takes a list of bowlling roles and makes a list of lists representing the frames of the bowling game. (in the example below we'll assume that a bowling game consists of 5 frames instead of 10).

```
(bowling-helper '(7 1 3 2 4 0 0 0 0 0)) ➔
                                        '((7 1)(3 2)(4 0)(0 0)(0 0))

(bowling-helper '(10 10 10 10 0 0)) ➔ '((10)(10)(10)(10)(0 0))


(bowling-helper '(10 10 10 10 10 10 10)) ➔
                                      '((10)(10)(10)(10)(10 10 10))
```

```
(define (bowling-helper bowling-game)
      (bowling-helper2 bowling-game 1))

(define (bowling-helper2 bowling-game frame)
  (cond
   ((equal? frame 5) (list bowling-game))
   ((is-strike? bowling-game)
      (cons '(10)
            (bowling-helper2
                  (remove-1-ball bowling-game)
                  (+ 1 frame))))
   (else
            (cons (get-2-ball-frame bowling-game)
            (bowling-helper2 (remove-2-balls bowling-game) (+
      1 frame))))))
```

```
(define (first-ball bowling-game)
```

```
  (car bowling-game))

(define (second-ball bowling-game)
  (cadr bowling-game))

(define (get-2-ball-frame bowling-game)
  (list (first-ball bowling-game) (second-ball bowling-game)))

(define (remove-1-ball bowling-game)
  (cdr bowling-game))

(define (remove-2-balls bowling-game)
  (cddr bowling-game))

(define (is-strike? bowling-game)
  (equal? (first-ball bowling-game) 10))
```