Your login name: `cs3-________`

Problem 1 (4 points, 10 minutes)

List the different ways that the `position` argument to `ttt` can be illegal. (Two ways count as different if they would be checked with different code.)

```
length 9, all chars either x or o or _

(define (legal? board)
   (and
      (= (count board) 9)
      (equal? board
         (keep (lambda (char) (member? char 'xo_)) board)) ) )

#x's = #o's and not an x win or #x's = #o's+1 and not an o win

(and
   ...
   (or
      (and
         (= (appearances 'x board) (appearances 'o board))
         (not (win? 'x board)) )
      (and
         (= (appearances 'x board) (+ 1 (appearances 'o board)))
         (not (win? 'o board)) )
```

Your login name: `cs3-________`

Problem 2 (8 points, 20 minutes)

Define a Scheme procedure named updated-hands that takes two arguments. The first is a list of two hands, each in the same format as hands used in the draw-another? homework assignment. The second is a rank. Updated-hands should return the result—again, a list of two hands—of transferring a card of the given rank from the second hand to the first. You may assume that the second hand contains at least one card of the given rank, and that hands do not contain duplicate cards. Here's an example:

```
expression                                 desired result

(updated-hands                             ((h10 ha hk c2) (d5 c6 d10 c8))
  '((ha hk c2)                             or
    (d5 c6 h10 d10 c8))                    ((d10 ha hk c2) (d5 c6 h10 c8))
  10 )
```

```
(define (transfer 2hands rank)
   (let ((card (find (cadr 2hands) rank)))
      (list
         (se card (car 2hands))
         (remove-once card (cadr 2hands)) ) ) )

(define (find hand rank)
   (first (keep (lambda (card) (equal? (bf card) rank)) hand)) )

remove-once appears on page 234

8 pts
-2 per bug, including using first for car and bf for cdr
finding a suitable card
transferring it to the other hand
removing it from the first hand
recreating the list
```

Your login name: `cs3-________`

Problem 3 (10 points, 32 minutes)

Part a

In the table below, list the arguments to each call to match-using-known-values that results from evaluating the expression

```
(match '(* a b) '(c a b))
```

You will not necessarily fill in all the blanks. Entries in the table should appear in the sequence that the corresponding calls to match-using-known-values are made (i.e. in the sequence you would see them if you traced match-using-known-values).

call number	*arguments*
1	`(* a b) '(c a b) '( )`
2	`'(a b) '( ) '( ) ; this call fails`
3	`'(a b) '(b) '( ) ; so does this one`
4	`'(a b) '(a b) '( )`
5	`'(b) '(b) '( )`
6	`'( ) '( ) '( ) ; success`
7	
8	
9	
10	

```
3 pts
-1 each distinguishable error; don't worry about the quotes
```

Part b

Devise a pattern that determines if its sentence argument contains the word “hundred”, associating the name “before-hundred” with whatever words occur in the sentence prior to the “hundred” and associating the name “after-hundred” with whatever words occur later.

```
'(*before-hundred hundred *after-hundred)

2 pts
-1 each error; don't worry about the quote
```

Part c

Fill in the blanks in the following procedure that, given as argument the result of matching a correct answer to part b with the name of an integer between 100 and 1000, returns the integer.

```
(define (numeric-value match-result)
   (+





         (* 100 (at-most-99-value                                         ))





         (at-most-99-value                                              ) ) )
```

Assume that a procedure at-most-99-value has been defined that, given a sentence naming a number less than 100, returns its numeric value.

```
(+
   (* 100 (at-most-99-value (sentence (second match-result))))
   (at-most-99-value (bl (bf (bf (bf match-result))))) )

5 pts
   correct use of sentence for first call to at-most-99-value: 1 pt
   correct access of # hundreds: 1 pt
   correct access of post-hundred: 3 pts, -1 for each missing procedure
```

Problem 4 (11 points, 35 minutes)

Consider a data base for a programming class, represented as a list, each element of which has the form

```
        (student-name project partner-list)
```

There is one element in the data base list for each student in the class. The element contains the student's name, the name of the project she worked on, and a list of the other students in the class with whom she worked in partnership on the project. This list does not include the student herself.

For this problem you are to write two versions of a procedure num-submissions that, given a data base and the name of a project, returns the number of submissions of that project. One project per partnership will be submitted, so each person in a partnership of size N will submit $1/N$ projects. Partnerships in this class may be of any size (unlike in CS 3 this semester).

Part a

Write a version of num-submissions that does not use higher-order procedures. (It may use recursion.) You may use helper procedures, as long as they too do not use higher-order procedures.

```
(define (num-submissions db proj)
  (if (null? db) 0
    (+ (contrib proj (car db)) (num proj (cdr db))) ) )

(define (contrib proj record)
  (if (equal? (project record) proj)
    (/ 1 (+ 1 (length (partners record))))
    0) )

5 pts
-2 per bug, e.g. off by 1
```

Part b

Write a version of num-submissions that does not use recursion. (It may use higher-order procedures.) You may use helper procedures, as long as their definitions do not use recursion.

```
(define (num-submissions db proj)
   (reduce +       ; apply or accumulate is ok
      (map
         (lambda (record) (/ 1 (+ 1 (length (partners record)))))
         (filter
            (lambda (record) (equal? (project record) proj))
            db) ) ) )

6 pts
-2 per bug, except -1 for using every instead of map and -1 for using keep
instead of filter
```

Problem 5 (10 points, 25 minutes)

Given below is a version of the flatten procedure from one of the lab activities. It doesn't work correctly.

```
; L is a list. Return the list of the atoms that occur in L,
; in the left-to-right order they appear in L.
(define (flatten L)
   (if (null? L) '( )
      (reduce append (map flatten L)) ) )
```

Part a

By filling in the blank below, provide a nonempty argument to flatten that does *not* cause an error when the resulting expression is evaluated.

```
(flatten ____________________________________________ )
```

```
a list whose elements are all empty lists
2 points
```

Part b

List the steps of the evaluation of (flatten '(A B)). (That is, imitate the Replacement Modeler.) If the result is an error, describe the error message and give the expression whose evaluation produced the error.

```
input to map isn't a list:
   (map flatten 'A) or (map flatten 'B)
2 points
```

Part c

Which of the following is a reason that flatten doesn't work correctly? (More than one may apply.)

a. The base case test (null? L) is incorrect.

b. The base case test is correct, but the empty list is the wrong result to return in that case.

c. The input passed to the recursive call is not what the procedure expects.

d. The input passed to the recursive call doesn't get closer to the base case.

e. Reduce and append are the wrong procedures to use with (map flatten L) to get the answer.

Explain your answer(s).

```
A: The base case should be testing for (word? L) and returning (list L).
C: The procedure wants a list and is getting a word.
3 points
```

Part d

Making as few changes as possible to flatten, fix it. Here's a copy of the code; indicate clearly what should be inserted, what should be deleted, and what should be changed.

```
      (define (flatten L)
         (if (null? L) '( )
            (reduce append (map flatten L)) ) )

(define (flatten L)
   (if (word? L) (list L)
      (reduce append (map flatten L)) ) )

3 points
```

Problem 6 (6 points; 23 minutes)

Consider the following higher-order procedure, which returns the result of combining some of the elements in its list argument into lists.

```
(define (some-paired L pred?)
   (cond
      ((null? L) L)                                        ; case 1
      ((null? (cdr L)) L)                                  ; case 2
      ((pred? (car L) (cadr L))                   ; case 3; #t means don't pair
       (cons (car L) (some-paired (cdr L) pred?)) )
      (else                                                ; case 4
       (cons
         (list (car L) (cadr L))
         (some-paired (cddr L) pred?) ) ) ) )
```

Part a

One may use some-paired to write a procedure grouped-into-frames that groups balls in a bowling game into frames, leaving strikes ungrouped. Grouped-into-frames, given the list

```
(9 1 0 10 10 10 6 2 7 3 8 2 10 9 0 2 7)
```

would thus return the list

```
((9 1) (0 10) 10 10 (6 2) (7 3) (8 2) 10 (9 0) (2 7))
```

Fill in the box below with a procedure that completes the definition of grouped-into-frames in terms of some-paired. Don't add anything else to the grouped-into-frames procedure; you may, however, supply a helper procedure. Assume for this part that the argument to grouped-into-frames is a list representing a game with a miss in the tenth frame.

```
(define (grouped-into-frames ball-list)
   (some-paired
      ball-list



) )
```

```
(lambda (ball1 ball2) (= ball1 10))

3 points
No credit without a procedure of two arguments.
-2: wrong ball used to compare
-1: != or < instead of =
```

-2: (lambda (x y) (> (+ x y) 10))
-1: putting a correct procedure definition (i.e. "(define (f rdigit1 rdigit2) ...") in the box without specifying a legal argument.

Part b (graded only if part a is essentially correct)

What would be the result of supplying the ball list below to your procedure from part a? (Note the strikes in the tenth frame.) Briefly explain your answer by identifying the cases in some-paired that handle the last three balls.

```
      (9 1 0 10 10 10 6 2 7 3 8 2 10 9 0 10 10 8)
```

```
((9 1) (0 10) 10 10 (6 2) (7 3) (8 2) 10 (9 0)
 10 10 8)
The strikes in the tenth frame are handled in case 3 and left untouched by
some-paired; then the (null? (cdr L)) in case 2 succeeds.

3 points, 1 for the answer, 2 for the explanation.

No credit if answer to part a is not of the form
   (lambda (x y) (op x 10))
```

Problem 7 (6 points, 20 minutes)

Consider a procedure named 1-extra? that, given two words as inputs, should return true exactly when the first word is the result of inserting a single letter into the second word. Examples:

expression	*desired result*
`(1-extra? 'HEAT 'HAT)`	true
`(1-extra? 'THAT 'HAT)`	true
`(1-extra? 'HATE 'HAT)`	true
`(1-extra? 'ABBC 'ABC)`	true
`(1-extra? 'AT 'HAT)`	#f
`(1-extra? 'HAT 'HAT)`	#f
`(1-extra? 'HATED 'HAT)`	#f
`(1-extra? 'CHEAT 'HAT)`	#f

A buggy version of 1-extra? appears below.

```
(define (1extra? big small)
   (= (num-matches big small) (- (count big) 1)) )

(define (num-matches big small)
   (cond
      ((empty? small) 0)
      ((equal? (first big) (first small))
       (+ 1 (num-matches (bf big) (bf small))))
      (else (num-matches (bf big) small)) ) )
```

Part a

Provide a call to 1-extra? for which it does not perform as intended, that is, it returns true or crashes when #f is the correct answer, or it returns #f or crashes when true is the correct answer.

Part b

Describe, as completely as possible, the set of words `wd` for which evaluating the expression

```
      (1-extra? wd 'HAT)
```

produces the correct answer. Briefly explain your answer.

```
Whenever wd contains the letters H, A, and T in order, it returns the correct
answer. Otherwise it crashes. That's because both recursive calls to num-
matches decrease the length of big, but the base case doesn't check whether big
is empty.
```

Problem 8 (5 points, 15 minutes)

Consider the following procedure, intended to return the result of inserting commas in the correct places (every third position starting from the right) in its nonnegative integer argument.

```
(define (with-commas n)
   (if (< n 1000)
      n
      (word (with-commas (quotient n 1000)) "," (remainder n 1000)) ) )
```

For example, (with-commas 12345678) should return the word 12,345,678. The procedure above, however, has a bug.

Part a

Describe, as completely as possible, the set of nonnegative integer arguments for which with-commas *does not work correctly.*

```
Any value in which a 0 would follow a comma.
2 pts
```

Part b

Fix the bug, changing the with-commas procedure as little as possible. You may define a helper procedure.

```
Replace the call to remainder by (with-leading-zeroes (remainder n 1000)):

(define (with-leading-zeroes lt1000)
   (cond
      ((< lt1000 10) (word "00" lt1000))
      ((< lt1000 100) (word "0" lt1000))
      (else lt1000) ) )

3 pts
```