CS3 Fall 05 – Midterm 2

Read and fill in this page now

Your name: ______________________

Your instructional login (e.g., cs3-ab): ______________________

Your lab section days and time: ______________________

Your lab T.A.: ______________________

Name of the person sitting to your left: ______________________

Name of the person sitting to your right: ______________________

Prob 0:		Prob 3:			
Prob 1:			Prob 4:		
Prob 2:		▨	Prob 5:		▨
Raw Total:		/50	**Scaled Total**:		/30

You have 70 minutes to finish this test, which should be reasonable; there will be approximately 10 additional minutes of leeway given. Your exam should contain 6 problems (numbered 0-5) on 12 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. If you have a question during the test, please come to the front or the side of the room to ask it.

Restrict yourself to scheme constructs that we have seen in this classs. (Basically, this excludes chapters 16 and up in Simply Scheme).

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. We believe we have provided more than enough space for your answers, so please don't try to fill it all up.

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

Relax!

Problem (1 point)

Put your login name on the top of each page.
Also make sure you have provided the information requested on the first page.

Problem (6 / 6 points). Remove-letter

Consider a procedure `remove-letter` that takes two inputs, a letter and a sentence, and returns the sentence with all occurrences of the letter removed. For example:

`(remove-letter 'e '(here is a sentence with e in it)`	➔	`(hr is a sntnc with "" in it)`
`(remove-letter 'e '(not any within))`	➔	`(not any within)`
`(remove-letter 'e '())`	➔	`()`

Part A: Write `remove-letter` without using any explicit recursion (i.e., use higher order functions instead)

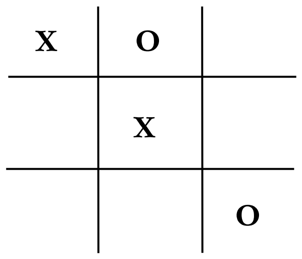
Problem continued

Part B: Write `remove-letter` without using higher-order functions (i.e., use recursion instead).

Problem (10 points): Not just a ticky-tack question

In tic-tac-toe, a pivot is an open square that identifies a winning move through the generation of a fork. In `ttt.scm`, the pivot procedure takes a sentence of triples and a player, and returns a sentence of pivots. The code in `ttt.scm` is reproduced in an appendix at the end of this exam.

For the board `b` equal to "`x o _ _ x _ _ _ o`", for example:



`(pivots (find-triples b) 'x)`	➔	`(4 7)`
`(pivots (find-triples b) 'o)`	➔	`()`

Rewrite `pivots` without using higher order procedures (i.e., using only recursion). You can use procedures defined in `ttt.scm` as long as those procedures don't use higher order functions. (You may use `appearances`).

Make sure to name your helper procedures and parameters well. You only need to comment when you think it necessary to help explain the intent of your procedure.

Here are some procedures you can use *without* writing them:

`keep-my-singles` takes a sentence of triples and a player and returns a sentence of triples that satisfy `my-single?` (that is, triples with two empty squares and one square filled by the player):

`(keep-my-singles (find-triples b) 'x)`	➔	`("4x6" x47 "3x7")`
`(keep-my-singles (find-triples b) 'o)`	➔	`("78o" "36o")`

`explode-all` takes a sentence of words and returns a sentence with each word "exploded" into single-letter words:

`(explode-all '(bob joe))`	➔	`(b o b j o e)`
`(explode-all '(25o 7o9))`	➔	`(2 5 o 7 o 9)`

Problem continued (space for your answer)

Problem (3 / 2 / 4 points): This is random

STk has a procedure `random` which is somewhat different than other procedures you have seen. Each time it is called, it returns a *different* random number. Random takes one argument, which specifies the upper bound on the random number that it will return:

`(random 10)`	➔	6	`(random 10)` *will return a number between 0 and 9. This time it was 6.*
`(random 10)`	➔	0	*Next time it was called, it returned 0. This might have been 6, though, since it is random!*
`(random 10)`	➔	3	*And again…*

Here is a buggy attempt to write a procedure to analyze at a bunch of random numbers:

```
;; check random runs (random val) n times, and returns the
;;   minimum and maximum values
(define (check-random val n)
   (check-random-helper val n 0 0))

;; this version doesn't work!  ...because random changes each time
(define (check-random-helper val n cur-min cur-max)
   (if (<= n 0)
      (se cur-min cur-max)
      (check-random-helper
            val
            (- n 1)
            (if (< (random val) cur-min)
               (random val)
               cur-min)
            (if (> (random val) cur-max)
               (random val)
               cur-max)
            )))
```

[problem continues on the next two pages]

Problem continued

Part A (3 points):. This version is buggy because `random` is called several times instead of once per each of the `n` cycles—remember, each call to random may return a different value! Fix check-random-helper *without* defining any additional procedures.

Part B (2 points). This version is buggy as well because, even with a correct fix to *Part A*, it always returns `0` as the minimum value of the set of random numbers. Fix this for *all possible* cases by modifying the `check-random` procedure (not the helper procedure) below:

Problem continued

Part C (4 points). Write `get-random-elements` which takes a sentence and returns the first `n` elements of the sentence, where `n` is a random number. For instance:

```
(get-random-elements '(this is a fine day)) can return either
      (),
      (this),
      (this is),
      (this is a),
      (this is a fine), or
      (this is a fine day) ,
```

Remember, `(random n)` returns an integer anywhere from `0` up to `n-1`. For instance,

`(random 5)` can return either `0`, `1`, `2`, `3`, or `4`.

You may use either recursion or higher-order procedures. You may use helper procedures.

Problem (3 / 6 points): It was a dark and mysterious recursion…

Consider the recursive procedure `gather` that takes a sentence of at least two single-character words (i.e., letters such as 'a', 'b', etc.):

```
;; sent-of-ltrs is a sentence of at least 2 words that are single
;;   letters
(define (gather sent-of-ltrs)
   (cond ((empty? sent-of-ltrs) '())
         ((empty? (bf sent-of-ltrs))
          (se (first sent-of-ltrs)))
         ((equal? (first (first sent-of-ltrs))
                  (first (bf sent-of-ltrs)))
          (gather (se (word (first sent-of-ltrs)
                            (first (bf sent-of-ltrs)))
                      (bf (bf sent-of-ltrs)))))
         (else
          (se (first sent-of-ltrs)
              (gather (bf sent-of-ltrs))))))
```

Part A (3 points). What will `(gather '(a b b b c d d))` return?

Part B (6 points). Write `gather-hof`, which behaves the same as `gather` but uses no explicit recursion.

Problem (9 points): Does money grow on tree recursions?

Consider a set of three coins: a penny, worth 1 cent; a nickle, worth 5 cents; and a dime, worth 10 cents. Write a procedure named `possible-amounts` which takes a number `n`, and returns a sentence of all the possible amounts that any `n` coins of these three types can make. For instance

`(possible-amounts 1)`	➔	`(1 5 10)`
`(possible-amounts 2)`	➔	`(2 6 11 10 15 20)` *(This includes two pennies, a penny and a nickel, a penny and a dime, two nickels, a nickel and a dime, and two dimes)*
`(possible-amounts 3)`	➔	`(3 7 12 11 16 21 15 20 25 30)`

Fill in the blanks to make the definition of `possible-amounts` work correctly:

```
(define *coin-amounts* ____________________________)

(define (possible-amounts n)
   (pa-helper *coin-amounts* n))


(define (pa-helper coins n)

   (cond ((<= n 1) ____________________)              ;; base case 1

         ((empty? coins) _________________________)   ;; base case 2

         (else (se (add-coin-to-every                 ;; recur case 1
                         (first coins)
                         (pa-helper coins (- n 1)))
                   (pa-helper                         ;; recur case 2

                         _______________________________________

                         __________________________________ )))))


;; add coin to each element of sent
(define (add-coin-to-every coin sent)
   (every (lambda (num)
             (+ coin num))
          sent))
```

Instructional Login ________________________

APPENDIX: the tic-tac-toe code from `ttt.scm`

```
(define (ttt position me)
  (ttt-choose (find-triples position) me))

(define (find-triples position)
  (every (lambda (comb) (substitute-triple comb position))
         '(123 456 789 147 258 369 159 357)))

(define (substitute-triple combination position)
  (accumulate word
            (every (lambda (square)
                      (substitute-letter square position))
                   combination) ))

(define (substitute-letter square position)
  (if (equal? '_ (item square position))
      square
      (item square position) ))

(define (ttt-choose triples me)
  (cond ((i-can-win? triples me))
        ((opponent-can-win? triples me))
        ((i-can-fork? triples me))
        ((i-can-advance? triples me))
        (else (best-free-square triples)) ))

(define (i-can-win? triples me)
  (choose-win
   (keep (lambda (triple) (my-pair? triple me))
         triples)))

(define (my-pair? triple me)
  (and (= (appearances me triple) 2)
       (= (appearances (opponent me) triple) 0)))

(define (opponent letter)
  (if (equal? letter 'x) 'o 'x))

(define (choose-win winning-triples)
  (if (empty? winning-triples)
      #f
      (keep number? (first winning-triples)) ))

(define (opponent-can-win? triples me)
  (i-can-win? triples (opponent me)) )

(define (i-can-fork? triples me)
  (first-if-any (pivots triples me)) )

(define (first-if-any sent)
  (if (empty? sent)
      #f
      (first sent) ))
```

Instructional Login ________________________

```
(define (pivots triples me)
  (repeated-numbers (keep (lambda (triple) (my-single? triple me))
                          triples)))

(define (my-single? triple me)
  (and (= (appearances me triple) 1)
       (= (appearances (opponent me) triple) 0)))

(define (repeated-numbers sent)
  (every first
         (keep (lambda (wd) (>= (count wd) 2))
               (sort-digits (accumulate word sent)) )))

(define (sort-digits number-word)
  (every (lambda (digit) (extract-digit digit number-word))
         '(1 2 3 4 5 6 7 8 9) ))

(define (extract-digit desired-digit wd)
  (keep (lambda (wd-digit) (equal? wd-digit desired-digit)) wd))

(define (i-can-advance? triples me)
  (best-move (keep (lambda (triple) (my-single? triple me)) triples)
             triples
             me))

(define (best-move my-triples all-triples me)
  (if (empty? my-triples)
      #f
      (best-square (first my-triples) all-triples me) ))

(define (best-square my-triple triples me)
  (best-square-helper (pivots triples (opponent me))
                      (keep number? my-triple)))

(define (best-square-helper opponent-pivots pair)
  (if (member? (first pair) opponent-pivots)
      (first pair)
      (last pair)))

(define (best-free-square triples)
  (first-choice (accumulate word triples)
                '(5 1 3 7 9 2 4 6 8)))

(define (first-choice possibilities preferences)
  (first (keep (lambda (square) (member? square possibilities))
               preferences)))
```