CS 3 Midterm 2

Spring 2006

Read and fill in this page now

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**:	

Official Use Only! No suggestions!

Problem (Pts)	**Your Score**
Q0 (1)	
Q1 (8)	
Q2 (9)	
Q3 (8)	
Q4a (7)	
Q4b (8)	
Q5a (6)	
Q5b (8)	
Raw Total (out of 55)	
Scaled Total (30)	

You have 80 minutes to finish this test, which should be reasonable. Your exam should contain 6 problems (numbered 0-5) on 8 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 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.

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

Good Luck!

Problem (1 point, 1 minute)

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

Problem (8 points) Tail recursion.

Write a tail-recursive definition for `n-matches`, which returns up to the first `n` matches between two sentences. A "match" occurs when the same word is at the same position in both sentences:

`(n-matches '(a b c d) '(a b c d) 3)`	➔	`(a b c)`
`(n-matches '(a x c) '(a b c d) 6)`	➔	`(a c)`
`(n-matches '(c b a) '(a b c) 3)`	➔	`(b)`
`(n-matches '(a b c) '(x y z) 2)`	➔	`()`

The numeric argument to `n-matches` will be 0 or greater. (Note that you will get partial credit if you write a non-tail-recursive solution to the problem.)

Problem (9 points) Higher-order election research

Write a higher-order procedure named `electoral-votes` which takes a predicate as its single argument. The procedure will sum up the 2008 electoral votes for states that satisfy the predicate.

`(electoral-votes california?)`	➔	55
`(electoral-votes blue-state?)`	➔	212
`(electoral-votes voted-for-bush-in-2004?)`	➔	286

The database of states and their electoral votes is in a global variable `*states*`, with the same format as in your third miniproject:

```
(ca55 me04 nj15 …)
```

The predicate takes the state's two-letter abbreviated name as its argument. You do not have to write these predicates; rather, you only need to write `electoral-votes` such that it works properly with any proper predicate.

Do not use any explicit recursion in your solution. Make sure you define and use accessor procedures where appropriate; you will lose some points if you don't.

Problem (8 points) A unique bug

Write the predicate `all-unique?`, which takes a sentence of any length and returns

- `#f`, if two or more words in the sentence are identical
- `#t`, otherwise

Do not use any explicit recursion. (That is, use higher order functions in your solution). Make sure to think about the full range of inputs.

Also, do not use the predefined procedures `dupls-removed` or `appearances`.

Problem (A: 7, B: 8 points). A troubling occurrence...

Recall the `occurs-in?` homework assignment from earlier in the semester. This predicate took two words as arguments, and returned `#t` if the first argument was contained somewhere in the second.

This question concerns a *semi-predicate* version of `occurs-in?`, named `occurs-in` (that is, without the "`?`"). A semi-predicate returns either `#f` or some scheme value that evaluates to true. In this case, when true `occurs-in` returns the portion of the second argument after and including the first argument:

`(occurs-in 'ab 'xabcab)`	➔	`abcab`
`(occurs-in 'ab 'nothere)`	➔	`#f`
`(occurs-in "" 'anything)`	➔	`anything`
`(occurs-in 'bb 'abbbc)`	➔	`bbbc`

Part A. Use the `occurs-in` semi-predicate to write a definition for `num-occurrences`, which takes the same arguments as `occurs-in` and returns the number of times that the first argument occurs within the second argument. (You can assume that the first argument will not be empty).

`(num-occurrences 'ab 'xabcab)`	➔	`2`
`(num-occurrences 'ab 'nothere)`	➔	`0`
`(occurs-in "" 'anything)`	➔	*not defined*
`(num-occurrences 'bb 'abbbc)`	➔	`2`

Make sure that you make <u>as few calls to `occurs-in` as possible</u> within your code. (As rationale, you could assume that `occurs-in` is a very slow procedure, and that if you call it too many times it will slow down your implementation of `num-occurrences` unacceptably. Do not rewrite portions of `occurs-in` to avoid using it, though!)

Part C: Your sometimes brilliant, sometimes completely-lost partner has written the following definition of the semi-predicate `occurs-in` by using the full predicate `occurs-in?` :

```
(define (occurs-in sub full)
   (if (occurs-in? sub full)
      (occurs-in-help sub (bf full) full)
      #f))

(define (occurs-in-help sub full prev-full)
   (if (occurs-in? sub full)
      (occurs-in-help sub (bf full) full)
      prev-full))
```

Briefly explain under which sets of inputs this procedure will:

1) return the correct value for `occurs-in`
2) return an incorrect value for `occurs-in`
3) crash or recurse infinitely (and, therefore, return no value).

Not all of these results may be possible.

Problem (A: 6, B: 8 points) Its for the kids

Kindergarteners spend a lot of time learning words of the form *consonant-vowel-consonant*, such as "cat" and "dog". This question will involve the procedure `make-kindergarten-words` (abbreviated `make-kw`), which takes a sentence of consonants and a sentence of vowels, and returns a sentence of **all** the possible kindergarten words that can be made using those letters. The ordering of the result sentence doesn't matter.

`(make-kw '(s t) '(a o))`	➔	`(sas sat sos sot tas tat tos tot)`
`(make-kw '() '(a o))`	➔	`()`

`(make-kw '(l n k t s)`
`'(a e i o u))`
	➔	`...` *`A really long sentence (225 words)!`*

Part A. Fill in the box below to complete the solution for `make-kw`. Do not use any additional helper procedures.

```
(define (make-kw consonants vowels)
   (every (lambda (c)
             (every (lambda (v)





                    )
                   vowels))
          consonants))
```

Part B. Complete the following recursive solution for `make-kw`, filling in the bodies for recursive cases 1-3 Don't use any higher order procedures.

```
(define (make-kw consonants vowels)
   (make-kw-help1 consonants vowels consonants))


(define (make-kw-help1 front vowel end)
   (cond ((or (empty? front)                             ;; base
cases
               (empty? vowel)
               (empty? end))
          '())

         ((and (word? front) (word? vowel) (word? end))   ;;
recursive
                                                          ;; case 1







          )

         ((and (word? front) (word? vowel))               ;;
recursive
                                                          ;; case 2







           )

         ((word? front)                                   ;;
recursive
                                                          ;; case 3







          )
         (else                                            ;;
recursive
          (se (make-kw-helper (first front) vowel end)    ;; case 4
              (make-kw-helper (bf front) vowel end))
          )
       ))
```