CS 3 Midterm 2

Fall 2006

Read and fill in this page now

Name:	
Instructional login (eg, cs3-ab):	
UCWISE login:	
Lab section (day and time):	
T.A. (-1 if you get the name wrong!):	
Name of the person sitting to your **left**:	
Name of the person sitting to your **right**:	

You have 80 minutes to finish this test, which should be reasonable. Your exam should contain 6 problems (numbered 0-5) and one appendix on 12 total pages. Note that problem 4 has three parts.

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!

Official Use Only! No suggestions!

(1 pt)	Prob 0	
(4)	Prob 1	
(9)	Prob 2	
(8)	Prob 3	
(8)	Prob 4a	
(6)	4b	
(10)	4c	
(2)	Prob 5a	
(6)	5b	
Raw Total (out of 54)		
Scaled Total (30)		

Problem 0. (1 point)

Put your login name on the top of each page.

Problem 1. One and only one letter different... (4 points)

The predicate `one-ltr-different?` takes two words, and returns `#t` if one and only one of the letters in the first word is different from the letter at the corresponding position in the second word. Otherwise, `one-ltr-different?` should return `#f`.

`(one-ltr-different? 'abcde 'abxde)`	➔	`#t`
`(one-ltr-different? 'abcde 'abxxe)`	➔	`#f`
`(one-ltr-different? 'abcde 'abcde)`	➔	`#f`
`(one-ltr-different? 'abcde 'bacde)`	➔	`#f`
`(one-ltr-different? 'abcde 'abxcde)`	➔	`#f`

Below is a buggy version of `one-ltr-different?`. Describe (precisely) what the inputs are that will return an incorrect value for this version.

```
(define (one-ltr-different? wd1 wd2)
  (cond ((and (empty? wd1) (empty? wd2))
         #t)
        ((or (empty? wd1) (empty? wd2))
         #f)
        ((not (equal? (first wd1) (first wd2)))
         (equal? (bf wd1) (bf wd2)))
        (else (one-ltr-different? (bf wd1) (bf wd2)))
        ))
```

Problem 2. From the beginning: a bunch of "easy" HOFs. (9 points).

In the following problems, you need to use HOFs to write procedures that mimic the functionality of `first`, `last`, `butfirst`, `butlast`, and `appearances`. You

- cannot use recursion or helper procedures
- cannot use any of `first`, `last`, `butfirst`, `butlast`, or `appearances;`
- cannot use the procedure `item`; and
- cannot go outside of the framework code given.

If you think a solution is impossible, you very well may be right: some of these are impossible! Write `IMPOSSIBLE` next to the problem to indicate that.

Don't get stuck on this problem: there is plenty more test after this...

```
;; Mimic first with the following procedure, if possible
(define (my-first sent)
  (accumulate
    (lambda (_________________)

     _____________________________________________________________)
    sent))


;; Mimic last with the following procedure, if possible.
(define (my-last sent)
  (accumulate
    (lambda (_________________)

     _____________________________________________________________)
    sent))


;; Mimic butfirst with the following procedure, if possible
(define (my-butfirst sent)   ;; mimicking butfirst
  (accumulate
    (lambda (_________________)


     _____________________________________________________________)
    sent))


;; Mimic butlast with the following procedure, if possible.
(define (my-butlast sent)
  (accumulate
    (lambda (_________________)


     _____________________________________________________________)
    sent))
```

Problem 2 continued.

```
;; Mimic butfirst with the following procedure, if possible
(define (my-butfirst sent)
  (keep
    (lambda (__________________)

       ____________________________________________________________)
    sent))
```

```
;; Mimic butlast with the following procedure, if possible.
(define (my-butlast sent)
  (keep
    (lambda (__________________)

       ____________________________________________________________)
    sent))
```

```
;; Mimic appearances with the following procedure, if possible.
(define (my-appearances item sent)

  (_____________________
     (keep
       (lambda (__________________)

          ____________________________________________________________)
       sent) ))
```

Problem 3. Recursion is like decoding words... (8 points)

Write a recursive procedure `decode-word` which takes a sentence of encoded letters and returns the word that they form when decoded. Decoding is done with a sentence `*codes*` which contains 26 codes, one for each alphabetical letter in order. For instance:

```
(define *codes*  '(12 3 4 1 2 14 ...twenty more codes here...)
(decode-word '(4 12 3)) ➔ cab
(decode-word '(14 12 1 2)) ➔ fade
```

The sentence `'(a b c d e f …)` is likely to prove useful in your solution. You can assume that all encoded letters given as input to `decode-word` exist inside `*codes*`. Do not use any higher order functions—rather, use recursion.

```
(define (decode-word coded-sent)
```

Problem 4. Moving in the second dimension (A: 8 points, B: 6 , C: 9 points)

Consider a higher order procedure `every2d` which takes a procedure and *two* sentences. The procedure must take two arguments, and `every2d` will build a sentence by calling the procedure with the corresponding values of each of the input sentences.

If the two sentences are of unequal length, `every2d` should return the word `UNEQUAL`.

`(every2d + '(1 2 3) '(100 200 300))`	➔	`(101 202 303)`
`(every2d word '(cs i fu) '(3 s n))`	➔	`(cs3 is fun)`
`(every2d word '(a long long sent) '(short))`	➔	`UNEQUAL`

Part A. Below is code for `every2d`, which consists of a single call to `e2d-helper`. Use the provided header to `e2d-helper`, and fill in the body as an accumulating recursion so that `every2d` works correctly.

Do not write any additional helper procedures.

```
(define (every2d proc sent1 sent2)
   (e2d-helper  proc  sent1  sent2 '() )
   )


(define (e2d-helper proc sent1 sent2 answer)
```

Part B. Below is a possibly buggy version of `prefix-values-removed`, from the "Roman Numerals" case study, written using `every2d`.

Recall, `prefix-values-removed` takes a `number-sent` with possible prefixes, and returns a sentence of numbers that when summed will be the arabic representation of the roman numeral in question. (The Roman Numerals case study code is in Appendix A).

Be sure to notice that extra zeros are added to the sentences that `every2d` is passed.

```
;; takes a sentence of digit values, possibly containing prefixes
(define (prefix-values-removed number-sent)
   (every2d (lambda (shifted orig)
              (if (< shifted orig)
                  (- orig shifted shifted)
                  orig)
              )
            (se 0 number-sent)
            (se number-sent 0)
            ))
```

Provide <u>three *good* test cases involving prefixes</u>. Include the return value, and comment on whether the return value is correct (i.e., correct in that the Roman Numerals code will work correctly when it uses this version of `prefix-values-removed`).

Your test cases should test as wide a range of different (possibly problematic) conditions as possible.

Part C. Write `decode-word` (described in problem 3) without using explicit recursion; rather, use only higher order procedures. *Do* use the HOF `every2d` (you can assume that you have a properly working version).

Again, the sentence `'(a b c d e f g ...)` is likely to prove useful.

```
(define (decode-word coded-sent)
```

Problem 5. Counting descendants. (A: 2, B: 6 points)

Consider the procedure `num-descendants` that returns the number of descendants a fish will make within a certain number of generations. The procedure takes the number of generations as its argument.

A descendant is a direct child or a descendant's child. The number of descendants in one generation would be the number of direct children made, the number in two generations would be the children and grandchildren, and so on. Do not count the original fish in this number!

This particular species of fish will make three children each year that it is between 2 years old and 6 years old, inclusive. The fish is too young to make children before that, and dies at the beginning of its 7th year. Therefore, `(num-descendants 1)` should return `15`.

The following is a buggy version of `get-descendants`:

```
;; buggy version
(define (num-descendants gen)
  (nd-help gen 0) )

(define (nd-help gen age)
  (cond ((= gen 0) 0)
        ((< age 2)
         (nd-help gen (+ age 1)))
        ((< age 6)
         (+ (nd-help (- gen 1) 0)
            (nd-help (- gen 1) 0)
            (nd-help (- gen 1) 0)
            ))
        ((> age 6) 1)
       ))
```

Part A: What will the result be, using the buggy version above, of `(num-descendants 1)`?

Part B. Fix the buggy version of `num-descendants` so that it works correctly.

Your solution should follow the general strategy of the buggy version—using tree recursion—rather than using a mathematical approach with the procedures `*`, `expt`, and so forth.

Appendix A: "Roman Numerals" case study code, First Version (Appendix A in the case study)

```
; Return the decimal value of the Roman numeral whose digits are
; contained in roman-numeral.
; Roman-numeral is assumed to contain only Roman digits.
; Sample call: (decimal-value 'xiv), which should return 14.
(define (decimal-value roman-numeral)
  (sum-of-all
    (prefix-values-removed
      (digit-values roman-numeral) ) ) )

; Return a sentence containing the decimal values of the Roman digits
; in roman-numeral.
; Roman-numeral is assumed to contain only Roman digits.
; Sample call: (digit-values 'xiv), which should return (10 1 5).
(define (digit-values roman-numeral)
  (if (empty? roman-numeral) '( )
    (se
        (decimal-digit-value (first roman-numeral))
        (digit-values (bf roman-numeral)) ) ) )

; Return the decimal value of the given Roman digit.
(define (decimal-digit-value roman-digit)
  (cond
    ((equal? roman-digit 'm) 1000)
    ((equal? roman-digit 'd)  500)
    ((equal? roman-digit 'c)  100)
    ((equal? roman-digit 'l)   50)
    ((equal? roman-digit 'x)   10)
    ((equal? roman-digit 'v)    5)
    ((equal? roman-digit 'i)    1) ) )

; Return the result of removing prefixes from number-sent.
; Number-sent is assumed to contain only positive numbers.
; A prefix is a number that is less than its successor in the sent.
; The prefix and its successor are replaced by the difference between
; the successor value and the prefix.
; Sample call: (prefix-values-removed '(10 1 5)), which should return
; (10 4).
(define (prefix-values-removed number-sent)
  (cond
    ((empty? number-sent) '( ))
    ((empty? (bf number-sent)) number-sent)
    ((and (empty? (bf (bf number-sent)))        ; length = 2?
          (>= (first number-sent) (first (bf number-sent))) )
          number-sent)
    ((not (starts-with-prefix? number-sent))
     (se
        (first number-sent)
        (prefix-values-removed (bf number-sent)) ) )
    ((starts-with-prefix? number-sent)
     (se
        (- (first (prefix-values-removed (bf number-sent))) (first number-sent) )
        (bf (prefix-values-removed (bf number-sent))) ) ) ) )
```

```
; Return true if the number-sent starts with a prefix, i.e. a number
;.that's less than the second value in the sentence.
; Number-sent is assumed to be of length at least 2 and to contain
; only positive numbers.
(define (starts-with-prefix? number-sent)
  (< (first number-sent) (first (bf number-sent))) )

; Return the sum of the values in number-sent.
; Number-sent is assumed to contain only positive numbers.
(define (sum-of-all number-sent)
  (if (empty? number-sent) 0
    (+ (first number-sent) (sum-of-all (bf number-sent)) ) ) )
```