CS3 MIDTERM 2

Fall 2007

Read this page and fill in the left table now. Don't turn the page yet.

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**:	

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

You have 90 minutes to finish this test, which should be reasonable. Your exam should contain 6 problems (numbered 0-5) on 9 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. (Basically, this excludes chapters 16 and up in Simply Scheme). You can always use helper procedures and procedures from other questions you've answered, unless the question specifically disallows it.

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. (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 1. A searching and a replacing... (A: 6, B: 8 points)

Consider a new procedure `replace` which takes a sentence of words, a single letter to find, and a letter (or word) to replace it with. The replacement happens in each of the words in the input sentence.

`(replace '(and away we went) 'a 'x)`	➔	`(xnd xwxy we went)`
`(replace '(and away we went) 'a "")`	➔	`(nd wy we went)`
`(replace '(and away we went) 'a 'joe)`	➔	`(joend joewjoey we went)`

Part A:

Write `replace` without using higher order functions. Rather, use recursion.
Do not use the procedure `position`..

Part B.

Write replace using no explicit recursion. Rather, use higher order functions. You can assume that the sentence, and each word in the sentence, will be of length 2 or greater. You can define helper procedures as necessary.

A reminder: the HOF `every` returns a sentence even when it is passed a word as input. This may not be appropriate for your usage. For instance,

`(every (lambda (ltr) ltr) 'joe)`	➔	`(j o e)`
`(every (lambda (ltr) (word ltr ltr)) 'joe)`	➔	`(jj oo ee)`

Problem 2. Let the real replacing begin (10 points)

Consider a procedure `word-swap`, which takes three arguments:

1. A word within which to do the replacing. This argument will never be empty.
2. A word to find, as many times as it occurs, with the first argument. This argument will never be empty.
3. A word to replace occurrences of the second argument (in the first argument) with.

`(word-swap 'ababc 'abc 'x)`	➔	`abx`
`(word-swap 'mississippi 'is "")`	➔	`mssippi`

Problem 3. Election analyses... (10 points)

Write the procedure `avg%-in-close-races` which takes as arguments

- a party (either `democrat` or `republican`) and
- a voting results sentence of the same format as used in miniproject #3.

`avg%-in-close-races` returns the average percent received by the party for states where the republican and democrat votes were within 5 percentage points of each other (that is, the difference is 5 percentage points or less):

`(avg%-in-close-races 'democrat '(ca3035 ny0515 or2925))`	➔	`30` *[(35+25)/2]*
`(avg%-in-close-races 'republican '(ca3035 ny0515 or2925))`	➔	`29.5` *[(30+29)/2]*

Assume there will be at least one state where percentage vote for the two parties was within 5 percentage points of each other.

The procedure `abs` may be useful: it takes a (possibly negative) number, and returns the corresponding positive number. For instance, `(abs -5)` returns `5`, while `(abs 5)` also returns `5`.

Some <u>important</u> points:

- Use proper data abstraction (accessors) when processing the results sentence. In fact, liberally use helper procedures to make your code readable.
- Do not use any explicit recursion in your solution. Use only higher order procedures.

```
(define (avg%-in-close-races party results)
```

Problem 4. Long may you run... (10 points)

Consider the procedure longest-run which takes a sentence and returns the length of the longest consecutive series of identical words.

`(longest-run '(a b b b c d d))`	➔	`3`
`(longest-run '(a a b a a a a ccccc d))`	➔	`4`
`(longest-run '())`	➔	`0`
`(longest-run '(a b c d e))`	➔	`1`

The solution on the next page is buggy. Identify and correct the bug(s) by

- underlining or otherwise identifying the area of the bug,
- describing briefly how the bug will manifest,
- giving an example of an input that would result in a buggy return value, and
- fixing the bug with the smallest amount of changes to the buggy code (written either alongside the buggy code or nearby.

Do not define additional helper procedures.

```
(define (longest-run sent)


   (lr-helper (bf sent) (first sent) 1 1)
   )



(define (lr-helper  sent  current-wd  current-length  longest-so-far)


  (cond ((empty? sent)


          longest-so-far)


        ((equal? (first sent) current-wd)


          (lr-helper (bf sent) current-wd
                     (+ 1 current-length) longest-so-far) )


        ((> current-length longest-so-far)


          (lr-helper (bf sent) (first sent) 0 current-length) )


        (else


          (lr-helper (bf sent) (first sent) 1 longest-so-far) )

        ))
```

Problem 5. An a-maze-ing tree recursion... (10 points)

The procedure `walk` is used to traverse a maze. A maze consists hallways that run either north/south or east/west; and locations, places in the maze where the hallways branch or turn.

- `walk` takes two arguments: a location `loc` and a direction `dir`. A location is a word that could be anything; a direction is one of `n`, `s`, `e`, and `w` (corresponding to the north, south, east, and west respectively).
- `walk` can return several different things:
  - **`deadend`** means walking in direction `dir` from location `loc` resulted in a deadend.
  - **`exit`** is returned when a exit is reached.
  - If a sentence is returned, walk reached a new location. The first word in the sentence is the new location. The remaining 2-4 words in the sentence are directions that can be taken from the new location. Note: one of the directions will be the *opposite* of the one that was originally taken, implying that taking it would return you to your previous location. Directions can be in any order.
  - **`ar-matey`**, which means you ran into a pirate and are done for. A violent `deadend`, really.

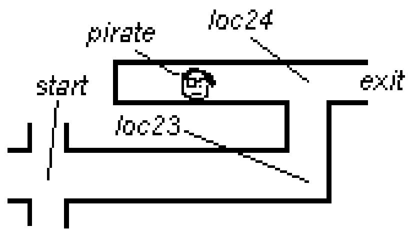
It is an error to call `walk` with a direction that can't be taken from the corresponding location.

You can use the procedure `opposite-dir` without writing it: it takes a single location and returns the opposite direction.

`(opposite-dir 'n)`	➔	`s`
`(opposite-dir 'e)`	➔	`w`

The maze has a special location `start` which is used to start the traversal of the maze. Consider the following possible snippet of the the maze and a series of calls to `walk`:

`(walk 'start 'e)`	➔	`(loc23 w n)`
`(walk 'loc23 'w)`	➔	`(start n s e w)`
`(walk 'loc23 'n)`	➔	`(loc24 w e s)`
`(walk 'loc24 'w)`	➔	`ar-matey`
`(walk 'loc24 'n)`	➔	*an error*
`(walk 'loc24 'e)`	➔	`exit`



(continued on next page)

You may or may not be able to exit the maze from the `start` location. Finish the definition for the procedure `solveable?`, which returns `#t` if it is possible to exit the maze. The maze doesn't contain any loops. Avoid looping infinitely by not returning to your previous location.

```
(define (solveable?)
  ;;your solution must call lead-to-exit?, defined below



  )

(define (lead-to-exit? loc dir)
  (let ((result (walk loc dir)))
    (cond ((equal? result 'exit) ____________)
          ((or (equal? result 'deadend)
               (equal? result 'ar-matey))
            ______________________________________)

          (else



            )
      ) ) )

;; additional helpers here
```