CS3 MIDTERM 2 FALL 2007
STANDARDS AND SOLUTIONS

Problem 1. A searching and a replacing... (A: 6 points ; 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`..

```
(define (replace sent find repl)
  (if (empty? sent)
      '()
      (se (replace-word (first sent) find repl)
          (replace (bf sent) find repl)) ))

(define (replace-word wd find repl)
  (cond  ((empty? wd)
          "")
         ((equal? (first wd) find)
          (word repl (replace-word (bf wd) find repl)) )
         (else
          (word (first wd)
               (replace-word (bf wd) find repl))) ) )

1 pt params
1 pt replace
1 pt replace-word base case
3 pts rec cases
```

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)`

```
(define (replace sent find repl)
  (every (lambda (wd)
           (replace-hof-word wd find repl))
         sent))

(define (replace-hof-word wd find repl)
  (accumulate word
              (every (lambda (ltr)
                       (if (equal? ltr find) repl ltr))
                     wd)))

1 pt param names
1 pt outer every
2 pt outer lambda (param, etc)
1 pt inner accum  (or 2 if no every)
1 pt inner every
2 pt inner lambda (param, etc)
```

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`

```
(define (word-replace source find rep)
  (cond ((empty? source)
         "")
        ((starts-with? source find)
         (word rep (word-replace (remove-front-elts (count find) source)
                                 find rep)) )
        (else
         (word (first source)
               (word-replace (bf source) find rep)))))


(define (starts-with? source start)
  (cond ((empty? start) #t)
        ((empty? source) #f)
        ((equal? (first source) (first start))
         (starts-with? (bf source) (bf start)))
        (else #f)))

(define (remove-front-elts n wd)
  (if (= n 0) wd (remove-front-elts (- n 1) (bf wd))))
```

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)


 (define (avg-pct-close party votes)
   (let ((close-contests (keep (lambda (state-result)
                          (<= (abs (- (vote-pct party state-result)
                                      (vote-pct (other-party party)
                                                state-result)))
                               5))
                          votes))
        )
     (/ (accumulate +
                (every (lambda (state-result)
                         (vote-pct party state-result) )
                       close-contests))
        (count close-contests)) ))
```

Problem 4. Long may you run... (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.

There were three errors here:

First, `longest-run` will error given an empty input, as tested in the third case above. An easy fix to this was to check specifically for an empty sentence before calling `lr-helper`; another fix involved sending `sent` directly to `lr-helper` (with changes to the other parameters, and changes to lr-helper).

Second, `lr-helper` will return an incorrect value if the longest run occurred at the end of the input sentence; it will return the longest run not including the last run. For instance, `(longest-run '(a b b c c c))` would return `2`. The easiest fix for this is in the first case of `lr-helper`, returning the maximum of `current-length` and `longest-so-far`.

The last error is in the third case: the recursive call to `lr-helper` should set `current-length` to `1` rather than `0`. Without this fix, runs following a longest so far (other than the first run) would be reported as too short by 1. For instance, `(longest-run '(a b b c c c c d))` would return `4`.

```
(define (longest-run sent)
   (lr-helper (bf sent) (first sent) 1 1)  ;; error 1
   )

(define (lr-helper  sent  current-wd  current-length  longest-so-far)
  (cond ((empty? sent)
          longest-so-far)  ;error 2. fix: (max current-length 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) ) ; error 3
        (else
          (lr-helper (bf sent) (first sent) 1 longest-so-far) )
        ))
```

Problem 5. An a-maze-ing tree recursion... (XXX 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: <u>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</u>. Directions can be in any order.
  - **`ar-matey`**, which means you ran into a pirate and are done for. A violent `deadend`, really.

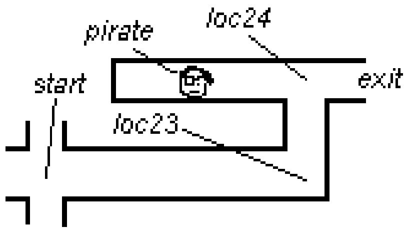
<u>It is an error to call `walk` with a direction that can't be taken from the corresponding location.</u>

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

   (or (lead-to-exit? 'start 'n)
       (lead-to-exit? 'start 'w)
       (lead-to-exit? 'start 'e)
       (lead-to-exit? 'start 's)))

  )

(define (lead-to-exit? loc dir)
  (let ((result (walk loc dir)))
    (cond ((equal? result 'exit) ________#t____)
          (or (equal? result 'deadend)
              (equal? result 'ar-matey))
           _______________________________#f_______)

          (else

(true-for-any? (lambda (newdir)
                     (lead-to-exit? (first result) newdir))
               (keep? (lambda (newdir)
                           (not (equal? (opposite-dir newdir)
                                        dir)))
                      (bf result)) ) )

           )
       ) ) )

;; additional helpers here

;; there are recursive solutions, and solutions that don't use helpers
(define (true-for-any? pred sent)
  (not (empty? (keep pred sent))))
```