The Doublets Game

Lewis Carroll, author of *Alice and Wonderland*, was a lover of word plays and puns. One puzzle he wrote about was called the game of *doublets*. (Perhaps you've played it before.) In classrooms around the world, we can imagine the following scene taking place:

> Alexander: Let's play doublets! Here are my two words: "horse" and "moose".
>
> Octavia: Ok... "horse" goes with "house" goes with "mouse" goes with "moose". Now you try. Here are my two words: "noise" and "posse".
>
> Alexander: ... This is too hard. Let's play Tic-Tac-Toe instead.

Formally, a word is a doublet of another word only if they're different in one letter. We can imagine a `doublet?` function that can tell us if two words are doublets of each other. For example,

```
> (doublet? 'noise 'poise)
#t
> (doublet? 'poise 'posse)
#t
> (doublet? 'water 'lager)
#f
> (doublet? 'game 'tame)
#t
```

Question 1 (**Easy**) Write a `doublet?` function that takes in two words, and tells us if they're doublets or not.

Question 2 (**Medium**) Octavia has written a small function called `make-doublet` that takes in a word and returns back a doublet of that word:

```
;; given a word wd, make-doublet gives back a ``doublet-fied'' word
(define (make-doublet wd)
  (let ((replacement-index
         (+ 1 (random (count wd)))))
    (make-doublet-helper wd replacement-index)))

;; given a word wd, and an index, make-doublet-helper changes a
;; particular letter in that word
(define (make-doublet-helper wd index)
  (if (= index 1)
      (word (mutate-letter (first wd))
            (bf wd))
      (word (first wd)
            (make-doublet-helper (bf wd) (- index 1)))))

;; given a letter, returns back another letter
(define (mutate-letter letter)
  (second (member letter '(l s y g c f v u h a e n o i p t
   b d j q w m r k x z))))

;; second returns the second item in a word or sentence
(define (second wd)
  (first (bf wd)))
```

- Right now, Octavia's program uses embedded recursion to construct the doublet, even with the helper. For practice, try rewriting this function so that it really is tail recursive.

- Also, although the program mostly works, there's one thing that frustrates Octavia: it occasionally breaks! Sometimes it works, and sometimes it doesn't. Octavia suspects it has something to do with the random letter choice, but the problem might lie elsewhere.

  Identify the problem in the program, and try correcting it. Hint: here are a few words that cause things to sometimes break: "zebra", "ostracized", "oz", "analyze", "doze", and "pez".

Question 3 (**Very Hard**) Alexander is tired of losing doublets to Octavia, so he decides he's going to use his superior programming prowess to double check that Octavia's not making the game impossible: he wants to write a program, `double-chain-exists?`, that takes the **start**ing word, **end**ing word, and a sentence of all the words in a **dictionary**. `double-chain-exists?` should says `#t` only if a doublet chain does exist between two words, and otherwise it should return `#f`.

```
;; Returns #t if some element in sent satisfies our predicate, fn
(define (some pred-fn sent)
  (cond ((empty? sent) #f)
        ((pred-fn (first sent)) #t)
        (else (some pred-fn (butfirst sent)))))

;; Returns a sentence of all the ``neighboring'' doublets of a word,
;; if we're given a dictionary to filter through.
(define (doublet-neighbors wd dictionary)
  (keep (lambda (dict-wd) (doublet? wd dict-wd)) dictionary))

;; Returns true if a doublet chain exists between start-wd and end-wd,
;; if we're restricted to using only words in our dictionary.
(define (doublet-chain-exists? start-wd end-wd dictionary)
  (let ((neighbors (doublet-neighbors start-wd dictionary)))
    (cond ((empty? neighbors) #f)
          ((member? end-wd neighbors) #t)
          (else (some (lambda (neighbor-wd)
                        (doublet-chain-exists? neighbor-wd end-wd
                                               dictionary))
                      neighbors)))))
```

Alexander's idea is to find a doublet chain recursively. If we're at the **start-wd** word, and if one of our "doublet-neighbors" is **end-wd**, then we're done! Otherwise, let's see if there's some neighbor that can build a doublet chain to the **end-wd**.

However, his program is buggy; it sometimes works, but most of the time, it goes into an infinite loop. For example:

```
> (doublet-chain-exists? 'noise 'posse '(posse poise noise))
#t
> (doublet-chain-exists? 'noise 'posse '(posse noise nails))
#f
> (doublet-chain-exists? 'noise 'posse '(posse noose noise))
                         ;; infinite loop!
> (doublet-chain-exists? 'horse 'moose '(horse house mouse moose))
                         ;; infinite loop!
```

Try to find out why it's getting stuck.

If you really have time on your hands, suggest ways of fixing the problem. (As a warning, this is WAY beyond core CS3 material: it touches on some issues in upper division CS!) So don't go crazy on this problem; for now, it's enough to *recognize* what's wrong.

There are several approaches to this problem, so don't feel as if there's just "one true way" to do it. Also, if you're getting stuck figuring why `doublet-chain-exists?` is getting stuck, feel free to talk to other people or your TA about this.