Word/List Processing

This exercise combines several of the topics that we've talked about recently: advanced recursion, higher-order functions, list manipulation, and sequential programming. The problems themselves shouldn't be too bad, but they do require juggling several concepts at once. We know that you're busy with the final project, but if you have spare time, feel free to try out these problems.

Question 1 (**Easy**) It's often very useful to write functions that present information that we can read easily. For example, let's say that we're writing a small word-processing program that represents a text document as a list of lines. For example, here's one example of such a document called `story`:

```
(define story '((h a r r y _ s p u n _ f a s t e r _ a n d _ f a s t e r)
                (e l b o w s _ t u c k e d _ t i g h t l y _ t o _ h i s)
                (s i d e s)))
```

A line is itself a list of single characters, and since it's a little inconvenient to use `" "` to put the spaces in there, let's use underscores instead. With this background in mind, let's try writing a function called `print-document` that takes this document and prints it out in human-readable format, one line at a time:

```
> (print-document story)
harry spun faster and faster
elbows tucked tightly to his
sides
()
```

To be more specific, `print-document` is a side effect function that takes a list of lines and prints it nicely, substituting underscores with blanks. Although we should ignore the return value, for completeness's sake, let's choose to have it return the empty list.

Write `print-document`. As you might notice, we're being really careful about using the word "list" instead of sentence, so keep in mind to avoid using the sentence functions for these exercises.

Question 2 (**Medium**) One reason why it's nice to represent a document as a list of lines is because we can perform operations on whole lines. For example, in many word processors, there's a justification option that lets us center our text or move it "flush right". Here's a procedure that, given a document and a line width, will justify the document flush right.

```
(define (flush-right-doc document width)
  (map (lambda (line)
         (flush-right-line line width)) document))

(define (flush-right-line line width)
  (if (< (length line) width)
      (flush-right-line (cons '_ line) width)
      line))
```

Let's see it in action:

```
> (flush-right-doc '((t h e r e _ s _ t a l k _ i n) (t h e _ c l a s s r o o m)
                     () (t o _ h u r t _ t h e y) (t r y _ a n d _ t r y)) 20)
((_ _ _ _ _ t h e r e _ s _ t a l k _ i n)
 (_ _ _ _ _ _ _ t h e _ c l a s s r o o m)
 (_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _)
 (_ _ _ _ _ _ _ _ t o _ h u r t _ t h e y)
 (_ _ _ _ _ _ _ _ _ t r y _ a n d _ t r y))
```

The idea of this problem is to review the different ways we can write these functions:

- The definition of flush-right-doc uses higher order procedures; try writing `flush-right-doc` in recursive form. Either embedded or tail recursion is fine.

- Since `flush-right-line` is tail recursive, see if there's a way of writing it in an embedded style.

Question 3 (**Hard**) There's another transformation we can do to a document that's probably hasn't been done in any other word processor. If we are silly enough, we can "rotate" a page of text on it's side using a function called `rotate-document` that works like this:

```
> (rotate-document '((t h e _ m a g i c)  (w o r d s _ a r e)
                     (s q u e m i s h)    (o s s i f r a g e))
((t w s o)
 (h o q s)
 (e r u s)
 (_ d e i)
 (m s m f)
 (a _ i r)
 (g a s a)
 (i r h g)
 (c e _ e))
```

`rotate-document` takes in a document, and puts it on its side. If necessary, it adds underscores so that each line is of the same width. We're written a procedure to do the "rotating" of a document:

```
;; ``rotate'' a document counterclockwise.
(define (buggy-rotate-document doc)
  (buggy-rotate-document-helper doc '()))

(define (buggy-rotate-document-helper old-doc new-doc)
  (if (all-null? old-doc)
      new-doc
      (let ((next-row (map car old-doc))
            (rest-of-doc (map cdr old-doc)))
        (buggy-rotate-document-helper rest-of-doc
                                      (list new-doc next-row)))))

(define (all-null? doc)
  (= (length doc) (length (filter null? doc))))
```

but as the names gently hint at, there's something slightly buggy here.

```
> (buggy-rotate-document '((a l l _ y o u r)
                           (b a s e _ a r e)
                           (b e l o n g _ _)
                           (t o _ u s _ _ _)))
(((((((((
        (a b b t))
       (l a e o))
      (l s l _))
     (_ e o u))
    (y _ n s))
   (o a g _))
  (u r _ _))
 (r e _ _))
> (buggy-rotate-document story)            ;; story comes from Question 1.
ERROR. car: expects argument of type <pair>; given ()
```

Experiment with these buggy procedures, and find out why they're giving buggy output.