Problem 4. (3 / 6 points): It was a dark and mysterious recursion...

Consider the recursive procedure gather that takes a sentence of at least two single-character words (i.e., letters such as 'a', 'b', etc.):

;; sent-of-ltrs is a sentence of at least 2 words that are single letters
(define (gather sent-of-ltrs)
  (cond ((empty? sent-of-ltrs) '())
        ((empty? (bf sent-of-ltrs))
         (se (first sent-of-ltrs)))
        ((equal? (first (first sent-of-ltrs))
                  (first (bf sent-of-ltrs)))
         (gather (se (word (first sent-of-ltrs)
                           (first (bf sent-of-ltrs)))
                  (bf (bf sent-of-ltrs))))
        (else
         (se (first sent-of-ltrs)
              (gather (bf sent-of-ltrs))))))

Part A (3 points). What will (gather '(a b b b c d d)) return?

Part B (6 points). Write gather-hof, which behaves the same as gather but uses no explicit recursion.
Problem 5. (9 points): Does money grow on tree recursions?

Consider a set of three coins: a penny, worth 1 cent; a nickle, worth 5 cents; and a dime, worth 10 cents. Write a procedure named possible-amounts which takes a number n, and returns a sentence of all the possible amounts that any n coins of these three types can make. For instance

<table>
<thead>
<tr>
<th>(possible-amounts 1)</th>
<th>(1 5 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(possible-amounts 2)</td>
<td>(2 6 11 10 15 20)</td>
</tr>
<tr>
<td>(This includes two pennies, a penny and a nickle, a penny and a dime, two nickels, a nickle and a dime, and two dimes)</td>
<td></td>
</tr>
<tr>
<td>(possible-amounts 3)</td>
<td>(3 7 12 11 16 21 15 20 25 30)</td>
</tr>
</tbody>
</table>

Fill in the blanks to make the definition of possible-amounts work correctly:

```
(define *coin-amounts* __________________________)

(define (possible-amounts n)
  (pa-helper *coin-amounts* n))

(define (pa-helper coins n)
  (cond ((<= n 1) ___________________)                 ;; base case 1
        ((empty? coins) ________________________)      ;; base case 2
        (else (se (add-coin-to-every                   ;; recur case 1
                    (first coins)
                    (pa-helper coins (- n 1)))
                  (pa-helper                           ;; recur case 2
                  ______________________________________
                  ______________________________________ ))))))

;; add coin to each element of sent
(define (add-coin-to-every coin sent)
  (every (lambda (num)
          (+ coin num))
         sent))
```