Problem 1. Remove consecutive duplicates (6 points, 11 mintues)

Consider a function `remove-conseq-dups` that takes a sentence and returns a sentence in which any occurrences of a word that follow that same word are removed. Note that the sentence may be of any length (e.g., empty).

For instance,

`(remove-conseq-dups '(the rain um um in spain um falls um um um mainly on on on um the plain))`	`(the rain um in spain um falls um mainly on um the plain)`
`(remove-conseq-dups '(a b a b c b))`	`(a b a b c b)`

Part A:

Write `remove-conseq-dups` without using higher order functions (i.e., using recursion). You may use helper procedures.

This is a somewhat standard recursion problem. Some points to consider: you will need to check both the first and second element of the sentence in the recursive case, so you need a base case that makes sure that the sentence has more than one element. And, even though you will check two elements into the list, you want to only shorten the list by one element when recursing: if you don't, you won't catch whether the second and third elements are duplicates.

```
(define (remove-conseq-dups sent)
   (cond ((empty? sent) '())
         ((empty? (bf sent))
          ;;only one element - can't be a duplicate
          sent)
         ((equal? (first sent)
                  (first (bf sent)))
          ;;aha, a duplicate.  Get rid of the
          ;;one
          (remove-conseq-dups (bf sent)))
         (else
          ;;not a dup, so keep the first element
          (se (first sent)
              (remove-conseq-dups (bf sent))))
         ))
```

This part was worth 3 points:

- Two base cases: 1 point total
- Recursive case when you want to throw away a duplicate: 1 point
- Recursive case when you don't want to throw away anything: 1 point

Some common mistakes included:

- Forgetting to check for one-element sentence base case (- _ pt.)
- Making the first recursive case

```
(se (first sent) (remove-conseq-dupls (bf (bf sent))))
```

since this fails if you have three consecutive duplicates (-1 pt)

Part B:

Write `remove-conseq-dups` without using explicit recursion (i.e., using higher order functions). You may use helper procedures.

The first thing to notice is that you cannot use `keep` or `every` here, because your function will need to look at more than one element at a time in order to check for duplication. `Accumulate` it is. The next thing to realize is that this is the kind of accumulate where you will need to keep the sentence around, building it up, as the accumulate checks pairs. You do this by returning a sentence from the function and receiving it in the "right" parameter of the successive call to the function. The first time that the function is called, however, you will get a word for that parameter.

Additionally, it is good form to make sure that your sentence has at least two elements before passing it to the accumulate. Remember that accumulate's behavior in those base cases can be strange. (In fact, the accumulate will fail on the empty sentence in this case, but work correctly on the 1-element sentence).

```
(define (remove-conseq-dups sent)
  (cond ((empty? sent) '())
        ((empty? (bf sent)) sent)
        (else
         (accumulate
           (lambda (left right)
             (if (word? right)
                 (if (equal? left right)
                     right
                     (se left right))
                 (if (equal? left (first right))
                     right
                     (se left right))))
           sent))))
```

The only ugliness is the nested `if`s inside the `lambda` (although they aren't *that* ugly), and that they do the same thing except for one part of the test. Surrounding `right` by a call to `sentence`, to insure that it is a sentence, is a method that some of you have suggested in lecture: certainly that works, but it seems wasteful to me. I prefer something like this:

```
...
(lambda (left right)
```

```
    (let ((right-elem (if (word? right)
                          right
                          (first-right))))
      (if (equal? left right-elem)
          right
          (se left right))))
```

Part B was worth 3 points
- using accumulate and lambda properly: 1 point
- knowing that there are two cases, one in which your lambda gets two words and one in which it gets a word and a sentence, and handling at least one of them properly: 1 point
- handling the other case properly: 1 point

Some common mistakes:
- forgetting that your lambda might take two words (-1 pt)
- forgetting that your lambda might take a word and a sentence (-1 pt)
- trying to do this with an every or a keep or both, which is completely impossible: you can't tell where in the sentence the word that you are looking at happens to be when you use these two procedures (-3 pts)

Problem 2. Rewrite `process-it` as a HOF (4 points, 8 minutes)

Rewrite `process-it` below using higher order functions (with no explicit recursion).

```
(define (process-it wd sent pred?)
   (cond ((empty? sent) '())
         ((pred? (first sent))
          (se (word wd (first sent))
              (process-it wd (bf sent) pred?)))
         (else
          (process-it wd (bf sent) pred?))))
```

The first task here, obviously, is to figure out what this recursive function is doing. The base case is the empty sentence: pretty standard. There are two recursive cases, depending on whether the first element of the sentence satisfies the `pred?` parameter. (For instance, if someone called `process-it` with `odd?`, then the two recursive cases would be selected for depending on whether the first element was odd or not).

If the first element does satisfy `pred?`, then `process-it` sticks `wd` on the front of that element, and keeps that new word as it processes the rest of the sentence. When the first element doesn't satisfy `pred?`, `process-it` just moves on to the rest of the sentence. So, this function both keeps only certain elements as well as does some processing to each element that it keeps. Looks like we'll need both an `every` and a `keep`:

```
(define (process-it wd sent pred?)
  (every (lambda (element)
           (word wd element))
         (keep pred? sent)))
```

The `(keep pred? sent)` provides the sentence that we pass to `every` – it makes sense to do the filtering first. The every needs a `lambda`, rather than some pre-defined function, because it needs to use the parameter `wd` that was provided to the call to `process-it`.

The problem was worth 4 points. You got 1 point for the `every`, 1 point for properly testing each element with `pred?` (this could be done either with a `keep` or an if expression inside the `every`), 1 point for a properly written `lambda`, and 1 point for adding the `wd` in front by writing `(word wd element)`

Problem 3. Growing buggy mountains (6 points, 11 minutes)

Consider a procedure mountains which takes a word as input, and returns a sentence containing words starting as the first letter of the input, then grow up to the full input word, and then back to the first letter.

`(mountains 'cs3)`	➔	`(c cs cs3 cs c)`

Part A

The table below contains versions of mountains that novice students submitted. (Not this brilliant class, of course, but some alternate universe). For each version, describe what the call `(mountains 'cs3)` will return in the box below. If a crash will occur, indicate why and where it happens, as specifically as possible. If the procedure will run infinitely, note that.

```
(define (mountains wd)
  (if (empty? wd)    '()
     (se (mountains (bl wd))
         wd
         (mountains (bl wd)))))
```

That solution looks reasonable, doesn't it? Actually, this is a tree recursion, because each recursive call with a non-base case will do two more recursive calls: a reasonably sized word will grow a very very large sentence.

The easiest way to get the answer is to trace out the call fully (I'll abbreviate `mt` for `mountains`):

```
(mountains 'cs3)
(se     (mt 'cs)        'cs3       (mt 'cs)    )
(se (mt 'c) 'cs (mt 'c) 'cs3 (mt 'c) 'cs (mt 'c)
(se (mt '()) 'c (mt '())
            'cs
          (mt '()) 'c (mt '())
                      'cs3  (mt '()) 'c (mt '())
```

```
                                          'cs
                                          (mt '()) 'c (mt '()))
(se '() 'c '() 'cs '() 'c '()
                          'cs3   '() 'c '() 'cs '() 'c '())
(c cs c cs3 c cs c)
```

This part was worth 1.5 points. You lost 0.5 points for a minor typo where the correct intent was clear.

```
(define (mountains wd)
  (se (mountains-helper wd)
      wd
      (reverse (mountains-helper wd))))

(define (mountains-helper wd)
  (if (empty? wd) '()
    (se (mountains-helper (bl wd)) wd)))
```

This takes a little more to understand, but gets closer to the right solution. You need to trace carefully to catch the bug, which most of you did.

```
(c cs cs3 cs3 cs3 cs c)
```

The "bug" is that mountains-helper was called each time on the full `wd`, rather than on the `(bf wd)`. The grading was the same as above.

Part B

Someone decides that `mountains` would look better if the sentence ended in the last letter of the input word, rather than the first:

`(better-mountains 'cs3)`	➔	`(c cs cs3 s3 3)`
`(better-mountains 'fred)`	➔	`(f fr fre fred red ed d)`

In fact, it does look better.

Write better-mountains.

The second buggy solution to *Part A* is a pretty good framework for this problem. But, well need to modify the helper to deal with the "better" part of `better-mountains` (i.e., that the word recedes towards the last letter after the full word, rather than the first letter). The easiest way is to write two different helpers:

```
(define (better-mountains wd)
   (se (mountains-helper (bl wd))
       wd
       (mountains-helper2 (bf wd))))

(define (mountains-helper wd)
```

```
  ... as in Part A...

(define (mountains-helper2 wd)
   (if (empty? wd) '()
       (se wd (mountains-helper2 (bf wd)))))
```

This problem was worth 3pts.

You lost -0.5 for minor "typo", if your intent was correct. Larger typos lost 1 point, such as `bf/bl` substitution or argument reversal if the answer otherwise demonstrates understanding of solution.

If the solution is just plain wrong, 1 pt for each component (main, helper1, helper2) that is correct (i.e., that would work in concert with other reasonable procedures)

Problem 4. Fill in the blanks (6 points, 10 minutes)

Fill in the blank to define a procedure `twice`. As input, `twice` takes a function *f* that takes a single input. As output, `twice` returns a procedure which takes a single input and applies *f* twice .

For instance, if `twice` were given a function that adds 3 to its input, `twice` would return a function that adds 6 to its input. If twice were given the function last, it would return a function that, when given a sentence, would return everything but the last two elements of the sentence.

```
(define (make-twice func)
    (lamda (x) (func (func x))))
```

```
(every (lambda (x)
         (cond ((negative? x) (* x -1))
               ((zero? x) 'zero)
               (else '())))
       '(2 6 -4 4 0 -9 0 9))

➔  (4 zero 9 zero)
```

Fill in the blank to define a procedure `make-member`, which returns a procedure that takes a word and returns true only if the word is in the sentence passed to `make-member`.

```
(define (make-member sent)
   (lambda (wd) (member? wd sent)))
```

Each part of the problem was worth two points:

Part 1:
- missing lambda -1.5 pt
- wrong function call -1 pt
- wrong lambda arguments -1 pt

Part 2:
- not handling a condition -1 pt

Part 3:
- incorrect lambda call -1 pt
- incorrect function call -1 pt

Problem 5. Modify the `pascal` program (6 points, 10 minutes)

The `pascal` procedure calculates the number in the corresponding cell of pascal's triangle, given valid row and column values:

```
(define (pascal col row)
   (cond ((equal? col 0) 1)
         ((equal? col row) 1)
         (else (+ (pascal col (- row 1))
                  (pascal (- col 1) (- row 1))))))
```

		columns (C)						
		0	1	2	3	4	5	...
rows (R)	0	1						...
	1	1	1					...
	2	1	2	1				...
	3	1	3	3	1			...
	4	1	4	6	4	1		...
	5	1	5	10	10	5	1	...
	...	...	...	...	...	...	...	...

Write a procedure `pascal-calls` that counts the number of recursive invocations that a particular call to `pascal` makes – that is, how many times `pascal` calls itself. `pascal-calls` will need to take a column and a row as input.

Provide three *good* test-cases (with return values) for valid invocations to `pascal-calls`.

Lets start with the base cases, since those will help with the design of `pascal-calls`. The procedure `pascal-calls` will have a similar structure to `pascal`, so we need to plan the test cases accordingly: we need to have a test case for the base cases and for the recursive case. So, something like:

```
(pascal-calls 0 3) ➔ 0   ;;base case col=0
(pascal-calls 4 4) ➔ 0   ;;base case col=row
(pascal-calls 2 4) ➔ 10  ;;recursive case
```

The return values are, as specified, the number of times that `pascal` would call itself if given those parameters. Calls that satisfy a base case immediately would have zero recursive calls. The non-base case calls will make two recursive calls, which each may or may not satisfy a base case (and therefore may not or may make two more recursive calls). You can figure this out by hand.

`Pascal-calls` will be a counting function, adding things up much like `pascal` does. `Pascal-calls` should count recursive calls, however; you can think of `pascal` as counting only base-case calls. (Essentially, `pascal` adds up the number of times a base-case, or a cell on the outside of the triangle, is reached during processing – it does this because it adds 1 for every base case, and nothing for every recursive case).

```
(define (pascal-calls col row)
   (cond ((equal? col 0) 0)
         ((equal? col row) 0)
         (else (+ 2
                  (pascal col (- row 1))
                  (pascal (- col 1) (- row 1))))))
```

Consider the relationship between the test cases above with the values that `pascal-calls` returns for each case. Many of you got the test case correct for the base case (i.e., returning 0), but still had the base-case in `pascal-calls` return 1!

3 points were given for the test cases: 0.5 for making having reasonable test case calls, and 0.5 for having correct return values for the base cases you did include.

3 points were given for the code to `pascal-calls`: 1 point for getting the return value in the base case correct (i.e., 0), 1 point for adding anything to the summation in the recursive case, and a final point for correctly adding 2 in the recursive case. Weird forms, including strange calls to `pascal` from within `pascal-calls`, or the use of a counter, lost anywhere from 1 to 3 points.

Many of you did try to use a counter in the fashion of an accumulating recursion, which isn't a bad idea on the face of it. However, it can't work: returning the incremented counter in the base case will make the summation far too large, and not returning the counter ever just won't work either! Try tracing your code to see this in action.