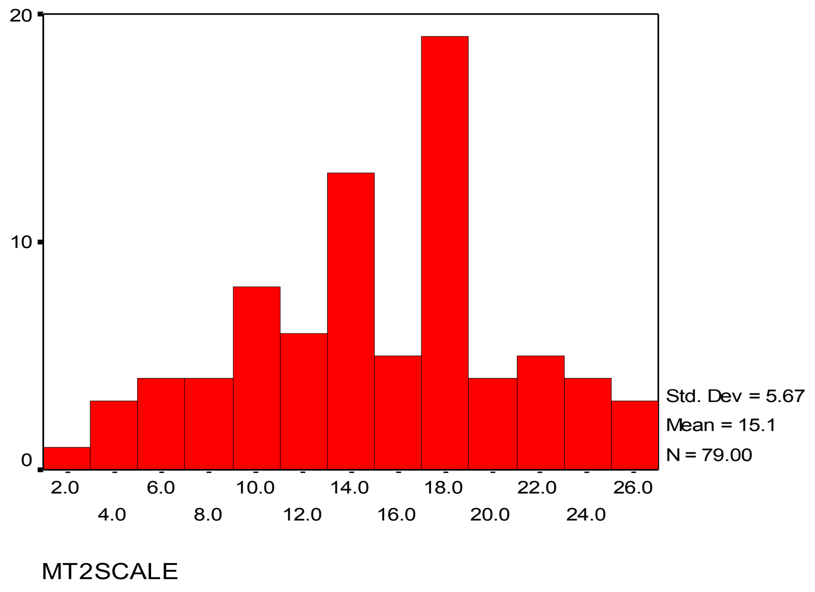
CS3 Spring 05 – Midterm 2

Standards and Solutions

This midterm was… difficult. More difficult that we as staff thought it would be. As such, grades were lower than we expected; consequently, steps will be taken to raise them when we calculate how much this midterm contributes to your final grade.

The minimum score, out of 28, was a 2; the highest score, a 26. The grade distribution looks like this:



Problem Left-handed accumulate (4 points)

Consider the higher order procedure `l2r-accumulate`, which works much like the `accumulate` that you are familiar with. The main difference, as you might expect, is that `l2r-accumulate` starts at the left of its input sentence, moving to the right.

`(l2r-accumulate + '(1 2 3 4))`	→	10
`(l2r-accumulate - '(1 2 3 4))`	→	-8
`(l2r-accumulate word '(abra ca da bra))`	→	abracadabra

Part A: Write `l2r-accumulate`. You can assume that it will only have to operate on sentences, rather than words. To make things easier, you can assume that the procedure that `l2r-accumulate` is given as the first parameter will only return words, rather than possibly sentences.

To make headway on this problem, you needed to be somewhat comfortable with what accumulate does. There are several ways to solve this problem; the easiest of which is a relatively simple recursion:

```
;; solution 1, proc must return a word, or a sent of length=1
(define (l2r-accumulate1 proc sent)
   (cond
      ((empty? sent) '())
      ((empty? (bf sent))
       (first sent))
      (else
       (l2r-accumulate1 proc (se (proc (first sent)
                                       (first (bf sent)))
                                 (bf (bf sent)))
                 ))))
```

Note that there are two base cases. The first deals with empty sentences, and will only be encountered when `l2r-accumulate` is called with an empty sentence. The second base case is necessary for the recursion to terminate properly.

The single recursive case for this solution should be understandable. Note, however, that if the `proc` returns a sentence of length 2 or greater, this recursion will never terminate. The problem did specify that you could assume that this would never happen.

Other solutions worked with any `procs` that returned sentences. Here is an accumulating recursive solution:

```
(define (l2r-accumulate2 proc sent)
   (if (empty? sent)
     '()
     (else (l2r-helper2 proc (first sent) (bf sent)))))

(define (l2r-helper2 proc so-far yet-to-accumulate)
   (if (empty? yet-to-accumulate)
      so-far
      (l2r-helper2 proc
                   (proc so-far (first yet-to-accumulate))
                   (bf yet-to-accumulate))))
```

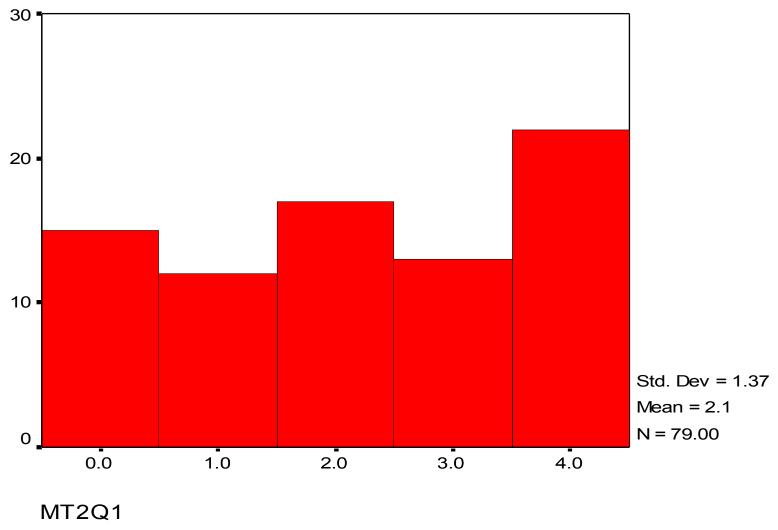
You could also write a solution similar to how accumulate was written, but have it use `bl` and `last`, rather than `bf` and `first` (this one is a little tricky to think about):

```
(define (l2r-accumulate3 proc sent)
  (cond ((empty? sent) '())
        ((empty? (bf sent)) (first sent))
        (else (proc (l2r-accumulate3 proc (bl sent))
                    (last sent)))))
```

Whew! There was one more (cheap) solution, involving `reverse` and `accumulate` which some of you tried. We meant to restrict you from using `reverse`, but we forgot, so this solution earned points *if done correctly*. You had to remember to reverse the arguments passed to the accumulate! (And, you would have to reverse the result sentence if `l2r-accumulate` were required to handle `procs` that returned sentences as well as words).

```
(define (l2r-accumulate4 proc sent)
   (if (empty? sent)
      '()
      (accumulate (lambda (l r) (proc r l))
                  (reverse sent))))
```

The distribution of scores on this question showed that many of you did reasonably well on this problem:



Grading: As far as base cases went, you were required to have a case for an empty input, as well as whatever base cases were required for *your* recursion to work. You did not explicitly have to account for a one-word sentence to be input. The answers given to this question were so varied that they were mostly graded on a case-by-case basis, but here were some "common" problems:

-4 if you unsuccessfully used some combination of higher-order functions
-2 if you ended up writing a right-to-left accumulate
-2 if you did not combine the result of a recursive call properly
-1 if you missed a necessary base case
-1 if your procedure skipped over words
-1 if your procedure did not properly use a result-so-far ½ is
-½ if you missed the empty-sentence base case
-½ if you returned the sent instead of (first sent)
-½ if the arguments to the input function were backwards (except, -2 with solution #4…)

Problem Certifying procedures (4 points)

`Every` can only take procedures (as its first argument) that return either a word or a sentence. Write the procedure `certify` which will ensure that the procedure given to an every will not cause an error due to its return value. `Certify` is used like this:

```
   (every (certify a-proc) '(a sentence to be mapped))
   (every (certify a-math-proc) '(1 2 3 4 5 6 7 8 9))
```

If the procedure passed to `certify` won't return a word or sentence, make it so that the word `"output-error"` is returned instead. For instance:

`(every number? '(friends 4 evah))`	➔	*ERROR*
`(every (certify number?) '(friends 4 evah))`	➔	`("certify-error" "certify-error" "certify-error")`

This problem threw many of you for a loop. The first thing to realize is that certify needs to return a procedure, because `every` requires that first parameter to be a procedure. Returning a procedure that already exists is easy in Scheme, as easy as returning a word or a sentence. Returning a procedure that is uniquely created each time, depending on the inputs, will necessarily involve using a `lambda` form! Remember those?

Certify takes a procedure as its argument; that should be obvious from the examples in the problem statement. Depending on the return values of that procedure, `certify` will either return a procedure that runs the original procedure on its arguments, or return a procedure that returns `"certify-error"`.

Several of you tried to make `certify` return an existing procedure – namely, the procedure that `certify` was given as a parameter – but ran into a problem:

```
(define (certify proc)
   (if (or (word? (proc 'no-idea))
           (sentence? (proc 'what-goes-here)))
        proc
        'certify-error))
```

The most obvious problem is what to pass as parameters to `proc` when you want to test its return value: because `certify` isn't being run when the call to `every` is being processed (as opposed to when it is being declared), you don't know what the inputs to `proc` are should be! Hence the silly arguments like, above, `'no-idea` and `'what-goes-here`. Something is very wrong, as some of you who wrote it this way probably realized.

Another obvious problem is that the return value `'certify-error` is not a procedure, so the every will bomb. A few of you got around this with a `(lambda (x) 'certify-error)` instead; clever, but this doesn't solve the first problem.

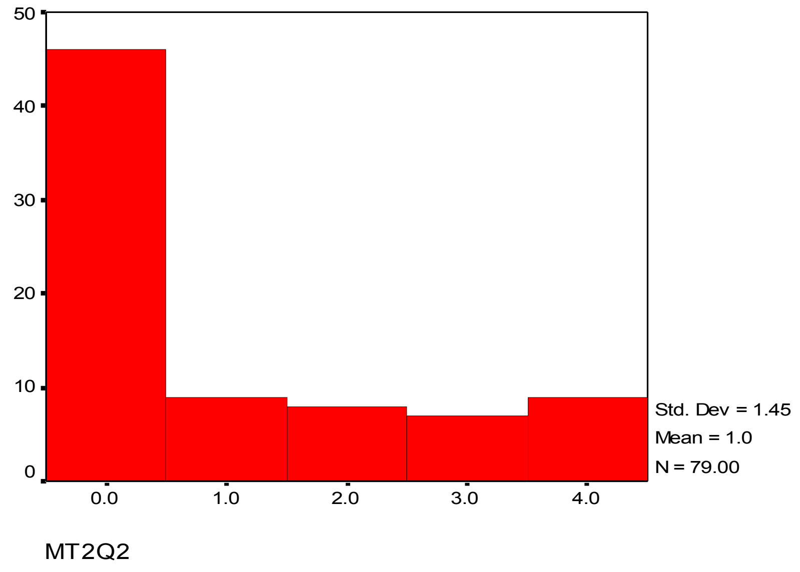
Here is the right solution, because the call to `certify` dynamically creates a procedure which does the certification every time it is run. Note the `lambda`:

```
(define (certify proc)
    (lambda (wd)
        (if (or (word? (proc wd))
                (sentence? (proc wd)))
            (proc wd)
            "output-error")))
```

Procedures like `certify` are called *wrapper* procedures, because they wrap other procedures and are able to do things like check the inputs, test whether the procedure will cause an error when run (something that Scheme can't directly do, but other programming languages can), modify the outputs, etc.

Grading: You lost two points if did not use lambda as the return value, one point if you didn't check both `word?` and `sentence?` completely, and one point if the return values (for the right or wrong cases) were incorrect.

Most of you had lots of trouble with this problem:



Problem (Re)cursing the tree (1/2/1/6 points)

This question concerns a data representation for a tree (i.e., the thing that grows in the ground, has a trunk, etc.). The representation of a tree is a sentence of branches, where each branch is a word. The first branch in the sentence is the trunk.

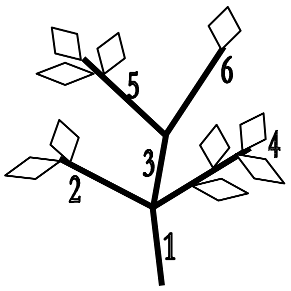
Each branch either has a certain number of leaves and no other branches coming off of it (called an "end-branch"), *or* has no leaves but connects to some number of other sub-branches. The other sub-branches are also contained in the tree data-structure.

An end-branch is represented by a word that starts with an "`e`" and ends with a number, which is the number of leaves on the end-branch. For instance, "`e12`" is a end-branch that has 12 leaves.

A non-end-branch starts with an "`x`", and ends with a series of branch positions in the tree sentence, with each position separated by a "`-`". For instance, the branch "`x3-4-5`" has three sub-branches which reside at position 3, 4, and 5 in the tree-sentence respectively.

Here is a "tree" that has 6 branches, including the trunk:

```
(x2-3-4 e2 x5-6 e4 e3 e1)
```



Part A: Write the predicate that tests whether a branch is an end-branch. Name this procedure and its parameter(s) properly.

> These first three parts of problem 3 were meant to be easy! For many they were, but it seemed like others spent way to much time on them. Data abstraction is a difficult concept, to be sure. For all these parts, you lost ½ a point for naming the procedure or its parameter improperly (a common mistake was to name the parameter `word` or `sent`).
>
> ```
> (define (end-branch? branch)
>     (equal? (first branch) 'e))
> ```
>
> You lost ½ a point for including the unnecessary `(if ... #t #f)` form here.

Part B: Write the selector procedure that returns the number of leaves on a branch. Name the procedure and its parameter properly.

> ```
> (define (number-of-leaves branch)
>    (if (end-branch? branch)
>        (bf branch)
>        0))
> ```
>
> Any attempts at recursion or HOFs lost a point. Forgetting about non-end-branches lost a point, since this was mentioned during the exam.

Part C: Write the selector procedure that returns the trunk for a given tree. Name the procedure and its parameter(s) properly.

```
(define (trunk tree)
   (first tree))
```

As per the last sentence in the first paragraph of this problem: there was no need to find the trunk via recursion or something like that! Some of you checked to see that the tree wasn't empty, which was a great idea (wish I had thought of it when writing the problem).

Part D: Write `count-all-leaves`

Assume that the procedure `sub-branches` has been written, which takes two arguments: a branch and a tree that contains it. `sub-branches` returns the sentence containing the sub-branches connected to the branch given as the first argument.

`(sub-branches 'x2-3-4 '(x2-3-4 e2 x5-6 e4 e3 e1))`	➔	`(e2 x5-6 e4)`
`(sub-branches 'e3 '(x2-3-4 e2 x5-6 e4 e3 e1))`	➔	`()`

Fill in the blanks on the procedures below, such that the procedure `count-all-leaves` will return the total number of leaves on a tree (You only need to count leaves on end-branches). Note that you **must** use only the selectors and predicates that you have defined earlier, and `sub-branches,` when accessing a branch or tree. **For instance, using a higher-order function to access a tree as a sentence is a data abstraction violation**!

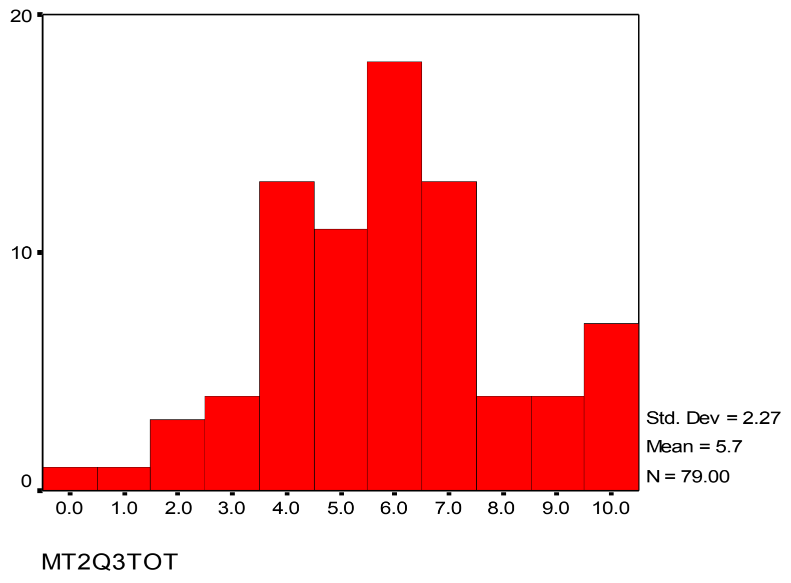
```
(define (count-all-leaves tree)
____(count-all-leaves-helper (trunk tree) tree))___


(define (count-all-leaves-helper branch tree)
   (if (end-branch? branch)
       ___(number-of-leaves branch)__;end-branch base case
       (accumulate                    ;recursive case
              ________+_____________
         (every
           (lambda (br) (count-all-leaves-helper br tree))
           (sub-branches branch tree)
               ))))
```

This is tree recursion. Although the tree is represented by a sentence, it would be a data-abstraction violation to access it via sentence selectors (like `first`, `bf`, or the higher order functions). Instead, the way you "traverse" this tree is by starting at the trunk and following all of the sub-branches, and all of their sub-branches, etc, until they terminate in an end-branch. The number of leaves for each end-branch is then added together.

This is recursive: for each sub-branch, you do the same thing as the trunk. One difference here is that you don't know how many recursive calls you are going to make at each step, since it depends on how many sub-branches there are. To that end, an `every` is used to make a recursive call for each sub-branch.

You received, basically, 1 point for each blank, except for the 2nd to last one, for which you receive a point for the lambda for and a point for the recursive call. Small data abstraction violations cost ½ a point (e.g., `(first tree)` in the first blank); larger ones cost more.

The distribution for question 3 (all four parts) looks like:



Problem Rewrite recursion using higher-order functions. (6 points)

Consider the following procedure:

```
(define (recurring-mystery sent cut)
   (cond ((empty? sent)
          "")
         ((special? (first sent))
          (word (first sent)
                cut
                (recurring-mystery (bf sent) cut)))
         (else
          (recurring-mystery (bf sent) cut))))
```

(For this procedure to run without errors, it may need additional procedures to be defined first). Write the procedure `hoffing-mystery`, without using any explicit recursion, such that it will return the same values as `recurring-mystery`, given the same input.

This procedure takes the elements of a `sent` that satisfy the predicate `special?`, puts `cut` in between each, and then collects them into a single word. The right answer looks like this:

```
(define (hoffing-mystery sent cut)
   (accumulate word
         (every (lambda (wd) (word wd cut))
                (keep special? sent))))
```

Several of you forgot to do the `accumulate`, which meant that your code would have returned a sentence rather than a word. We took off 2 points for this error.

Some of you had some problems using `every` and `accumulate`:

```
(define (hoffing-mystery sent cut)
   (accumulate (lambda (wd) (word wd cut))
                (keep special? sent))))
```

We took off 3 points for this error. The `lambda` is doing what should have been done by the `every`, and the `lambda` to `accumulate` should take in two arguments. Another common error:

```
(define (hoffing-mystery sent cut)
   (accumulate (lambda (x y) (word y cut))
                (keep special? sent)))

(define (hoffing-mystery sent cut)
   (accumulate (lambda (x y) (if (special? y) (word y cut)))
               sent))
```

We took off 3 points for these errors as well. In the both cases, we are only checking the accumulated `y` value for `special`-ness, and completely ignored the `x` value. Some had the same error, but evaluated the `x` instead of the `y`.

Another problem with the use of `accumulate`:

```
(define (hoffing-mystery sent cut)
   (accumulate (lambda (wd1 wd2) (if (word? wd2)
                                     (word wd2 cut) …))
                (keep special? sent))))
```

We took off 1 point for this error. The case `(word? wd2)` is always going to return true because, well, it is always a word. That is the return value of `hoffing-mystery`! We did something similar to this in lab, such as `position` form `diagonal`: checking if the second argument to `lambda` is a word. However, that was only when the accumulated value is a sentence in the end, and `(word? wd2)` only applied to the first call to `lambda`.

A last common problem with `accumulate we saw`:

```
(define (hoffing-mystery sent cut)
   (accumulate (lambda (wd1 wd2) (word wd2 cut wd1))
               (keep special? sent))))
```

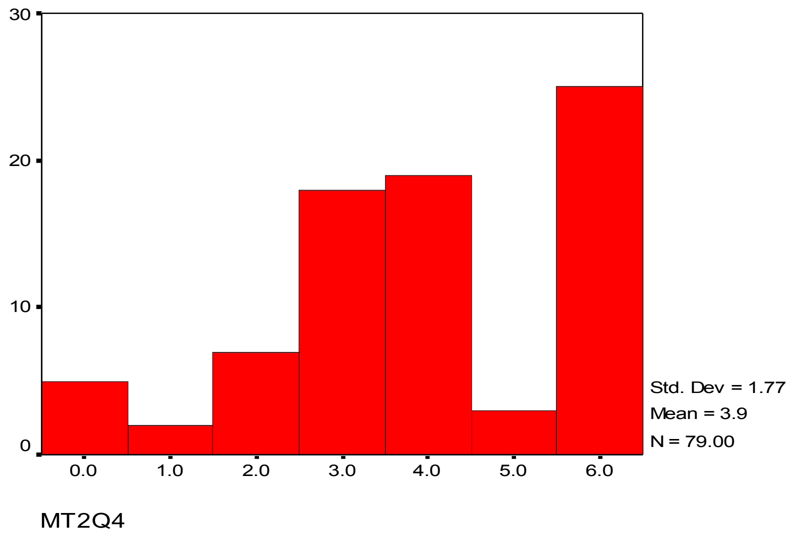
We took off 1 point for this error. The obvious error is the order of `wd1` and `wd2` should be switched. The more important is that the last (rightmost) "`special`" word does not have a `cut` included at the end of it.

Some of you passed in just sent as the second argument to `every` or `accumulate`, and handled the `special?` inside the `lambda`. This is how the `lambda` looked like:

```
(define (hoffing-mystery sent cut)
   (accumulate word
         (every (lambda (wd) (if (special? wd)
                                 (word wd cut)
                                wd)
                sent)))
```

We took off 1 point for this error. If the word is not `special`, then it should not be there at all! It is not just that it does not have a `cut` attached to it, we should discard it completely. Try `'()`, or "" which will appear in what `every` returns, but the empty words will disappear when `(accumulate word …)` is called.

The distribution of scores looks like:



Problem Rewrite a higher-order function as recursion (2/5 points)

Consider the following function:

```
(define (hof-of-horror sent)
   (every (lambda (v)
              (every (lambda (nv) (word v nv))
                     (keep (lambda (wd) (not (vowel? wd)))
                           sent)) )
          (keep vowel? sent)))
```

Part (A)

What does `(hof-of-horror '(a b c d e))` return?

OK, this problem was difficult. There is a lot going on here, but most of you did a good job understanding what `hof-of-horror` was doing: prepending each vowel in the input sentence onto every non-vowel. The answer to part A, then:

```
(ab ac ad eb ec ed)
```

Part (B)

Write the function `recursion-of-horror` (which you can abbreviate as `roh`) so that it will return the same values as `hof-of-horror`, given the same inputs. For `roh` and helpers, do not use `keep`, `every`, `accumulate`, `repeated`, or `lambda` forms.

There were several ways to tackle this problem. The first thing to do, though, was to call a helper procedure that got the parameter `sent` twice:

```
(define (roh sent)
  (roh-helper sent sent))
```

Some of you created simple recursions to duplicate what the `keep` originally did: strip out all of the non-vowels in the first sentence, and all of the non-vowels in the second (changing the second line above to `(roh-helper (filter-vowels sent) (filter-non-vowels sent))`). Others did the filtering inside the recursion that implemented, in essence, `prepend-every`. This second case looked like:

```
;recurse down the possible vowels, calling roh-helper2 on each
(define (roh-helper v-sent nv-sent)
   (cond ((empty? v-sent) '())
         ((vowel? (first v-sent))
          (se (roh-helper2 (first v-sent) nv-sent)
              (roh-helper (bf v-sent) nv-sent)))
         (else (roh-helper (bf v-sent) nv-sent))))

;recurse down the possible non-vowels, prepending the
; current vowel onto each non-vowel
(define (roh-helper2 v nv-sent)
   (cond ((empty? nv-sent) '())
         ((not (vowel? (first nv-sent)))
          (se (word v (first nv-sent))
              (roh-helper2 v (bf nv-sent))))
         (else (roh-helper2 v (bf nv-sent)))))
```

If the filtering was done outside of these helpers, there was a single recursive case in each.

Some of you created a single helper procedure in the same way as `thoroughly-reversed` in the lab materials: have two different sets of recursive cases depending on whether you are processing the vowel-sentence or the current vowel.

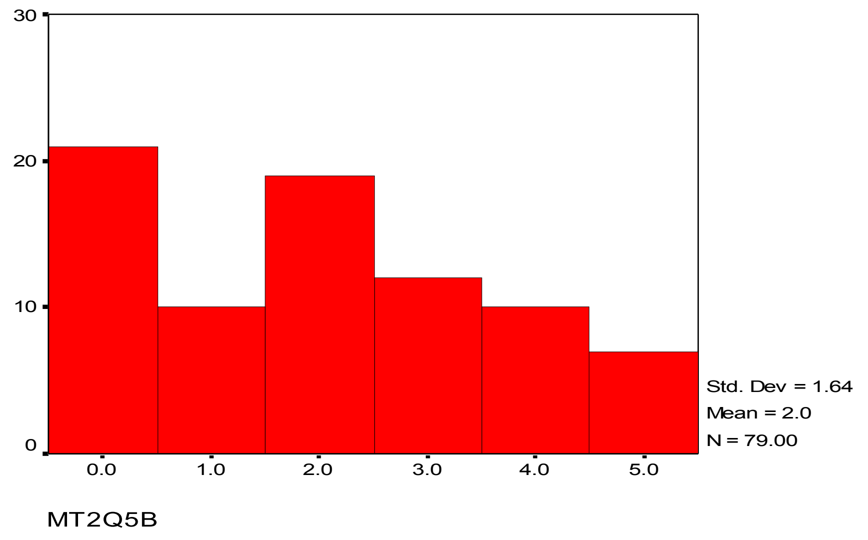
Grading: For part A, you lost one point if you only did half, or had the sentence reversed. With other mistakes, you lost both points.

For the part B, there were basically three issues:

- separate vowels and consonants (1 point each)
- prepend-every (2 points)
- coordinating all of this (1 point)

In general, mistakes were one point each. Things like throwing away a consonant before combining it with all vowels lost two points. Pairing up the first vowel and first consonant, second vowel and second consonant, etc., lost two points. Both of these were relatively common mistakes. Pairing up two letters only if they happen to be next to each other lost three points, but this was a less common mistake.

Another common mistake was not knowing the difference between an empty sentence (it looks like `()` and is a sentence) and an empty word (it looks like `""` and is a word). The distribution of scores for both parts together looks like:


Std. Dev = 1.64
Mean = 2.0
N = 79.00
MT2Q5B