Problem 1 (6 points, 10 minutes): Make the expressions correct

Add parentheses and procedures in the underlined areas to make these expressions return the value specified. Do not define any new procedures. Be careful with parentheses and quotes!

You can leave the underlined areas blank, but all the explicit text must be included in your expression—you can't just leave something out. If it is impossible to create an expression that returns the value specified, write IMPOSSIBLE somewhere (to the right of the return value, say).

Remember, this question was mis-worded: you could also add quoted words and sentences in the blanks below. This allowed for a solution to [4] below.

1]
```
(_bl (bl (se_ 'scheme 'is 'a 'state 'of 'mind ____) )____)
➔ (scheme is a state)
```

2]
```
(sentence    a _____ (item 3 ____'(state of mind)___))
➔ (a mind)
```

3]
```
(__item 8 (first___ '(expression)____)________)
➔ i
```

4]
```
(_word (first____ '(go cal bears) _) 'cal  'bears___)
➔ gocalbears
```

5]
```
(count ____""_____)
➔ 0
```

6]
```
(if (or #f (and _________________ #f))
   'yep 'nope)
➔ yep                IMPOSSIBLE
```

7]
```
(define (what-the sent)
   (if (empty? sent)
       ""
       (word "-um-" (first sent) (what-the (bf sent)))))
 (what-the _____'( I like "" cheese )________)
➔ "-um-i-um-like-um--um-cheese"
```

Part [5] also works with `'()`, which many of you used.
Some of you came up with other solutions for part [7] that none of us thought of, e.g.:
```
  (what the '(I like -um-cheese))
```

Each of these problems was worth one point, but only six points were possible. Each problem was graded all-or-nothing (i.e., no partial credit).

Problem 2 (5 points, 10 minutes) : Validating dates for `day-span`

Write a procedure called `day-span-validated` which takes the same arguments as `day-span` and returns the same value as `day-span` when the argument dates are valid (i.e., legal). If either of the dates is invalid, however, `day-span-validated` should return `(invalid date)`.

For instance:

`(day-span-validated '(february 30) '(june 3))`	➔	`(invalid date)`
`(day-span-validated '(june 4) '(may 15))`	➔	`-19`
`(day-span-validated '(january 19) '(flugalmonth 5))`	➔	`(invalid date)`
`(day-span-validated '(january three) '(may five))`	➔	`(invalid date)`

You can assume that each argument to `day-span-validated` is a sentence containing two words. You can (and should) use procedures from the "Difference Between Dates" case study, part I, that will help you. This code is included in an appendix at the end of the exam. In particular, consider the procedures `day-span`, `days-in-month`, `month-name`, and `date-in-month`. Assume that this is <u>not</u> a leap year.

There were three basic things you had to do for this problem:

1) make sure the dates are correct
2) call day-span correctly, if the dates are valid
3) correctly return `(invalid date)` if the dates are invalid
4) use procedures from the Difference Between Dates case study

Parts 2 and 3 are pretty simple. We took off 1 point each if they were missing.

Part 4 requires that you use `date-in-month` instead of `last` when you want the day of the month, `month-name` instead of `first` if you want the name of the month, `days-in-month` if you want to know how many days are in a month, and `day-span` if you want to do day-span. If you missed some of this, you lost a point. Many people wrote something like:

```
((and (member? (month-name date) '(september april june november))
      (> (date-in-month date) 30))
 #f)
```

and so on for 31-day months and again for February. You could and should have done this with one line:

```
((> (date-in-month date) (days-in-month (month-name date)))
 #f)
```

Part 1 is more complicated. There are at least four things you should do.

1) Make sure the name of the month is valid
2) Make sure the day is a number
3) Make sure the day is > 0
4) make sure the day is < `(days-in-month (month-name date))`

If you missed one of these, you lost a point. If you checked to see if the day was a number only after you compared it to other numbers, you lost half a point. You can't do < or > on something that's not a number, after all.

Several people really messed up their conds or ifs. Every if needs three parts: a test, something you do if the test returns true, and something you do if the test returns false. Putting a bunch of ifs together in a column will not make them work like a cond. Mistakes like these were worth between 1 and 2 points, depending on how confused the ifs were.

There were several cond mistakes. First, some people did something like

```
(cond ((and (valid-month? date1)(valid-month? date2))
        (day-span date1 date2))
       ((and (number? (date-in-month date1))
             (number? (date-in-month date2)))
        (day-span date1 date2))
       ((and (> (date-in-month date1) 0)
             (> (date-in-month date2)))
        (day-span date1 date2))
       ((and (<= (date-in-month date1)
                 (days-in-month (month-name date1))
             (<= (date-in-month date1)
                 (days-in-month (month-name date1)))
        (day-span date1 date2))
       (else '(invalid date)))
```

This will call day-span if the dates pass any one of those four tests. Making this mistake cost people two points. Another mistake was to leave off one set of parentheses from every cond line:

```
(cond (not (and (valid-month? date1) (valid-month? date2)))
      #f
      ...
```

This mistake cost people 1.5 points.

Problem 3 (6 points, 10 minutes): Data Abstraction in School

You are part of a team working on a project to create, among other things, a database system for schools. One of your tasks is to implement the school datatype in Scheme. This question does not require recursion.

In the project, a *school* will have three pieces of information:

- A short name, which will be a single word,
- A full name, which will be one or more words,
- Enrollment, a number

For instance, here are some possible *schools*:

full name	*short name*	*enrollment*
University of California Berkeley	Cal	23000
Rice	Rice	2822
California Institute of Technology	Caltech	900

This problem confused many people, but you have seen constructors and selectors in lab, lecture, and the book (look it up!). The Scheme syntax of the solutions was hardly difficult, which may have thrown some of you. The concept of data abstraction is a difficult one, but something important for this course!

Part A
Make a constructor procedure for the *school* datatype. Choose a reasonable name for the procedure, and include good comments. (You will, of course, need to decide how the data is stored; there is more than one correct way to do this.) Include a call to your constructor.

A solution:

```
(define (make-school fname sname enroll)
   (se sname enroll fname))

(make-school '(university of california Berkeley) 'cal 23000)
➔ (cal 23000 university of california berkeley
```

Other orderings of arguments and packaging up of the sentence were possible. Packaging up the three pieces of information into a word, while technically possible, would have required recursion in the selectors.

You lost 0.5 points for forgetting to provide a call; 0.5 points for a poor name. Bad or missing comments lost no points. Using selectors or including wrong arguments lost between 1 and 2 points, depending on the rest of the logic.

Part B
Make the three selector (accessor) procedures, with reasonable names and comments. DO NOT use recursion when writing these.

Using the above constructor:

```
(define (full-name sch) (bf (bf sch)) )
(define (short-name sch) (first sch) )
(define (enrollment sch) (first (bf sch)) )
```

Not writing selectors consistent with your constructor lost 0.5 points each (or 1 point from the constructor, depending). Other problems were worth between 0.5 to 1 point for each selector.

Part C
What is a reasonable name of the boolean test procedure for this datatype? (That is, the procedure that somebody can use to test whether a Scheme value is a *school* or not.) Don't write this procedure, just name it.

The best name is `school?` , as with other predicates. Not having the question mark lost 1 point; a name that didn't imply "is this a school" lost 1 point [the majority of you got this question].

Problem 4 (5 points, 9 minutes): Argue

Write a function called `argue-response` which takes a sentence and returns a sentence in which the meaning is reversed. Perhaps the easiest way to explain how it reverses meaning is through some examples:

`(argue-response '(I hate the opera))`	➔	`(you love the opera)`
`(argue-response '(you love cookies))`	➔	`(I hate cookies)`
`(argue-response '(you hate everything that I cook))`	➔	`(I love everything that you cook)`

If the first two words in the sentence are not of the form you've seen, `argue-response` takes a different tack, adding `(all you ever say is)` to the input:

`(argue-response '(lets have dinner))`	➔	`(all you ever say is lets have dinner)`
`(argue-response '(what))`	➔	`(all you ever say is what)`

The third example in the first set of test cases above contained an unfortunate error: the second to last "I" in the argument was transformed to a "you" by `argue-response`. Only a few of you caught this: solving for it required recursion. You were not taken off for not doing this recursion; including a recursive solution gave you some leeway to miss a point elsewhere (which no one did).

Part A:
Fill in the blanks below, assuming that the code implied by *[…]* is written correctly.

```
(define (argue-response remark)
  (cond
    ((starts-with? remark '(i hate))
      ______(se 'you 'love (bl (bl remark)))____________
      )
    ((starts-with? remark '(i love))    […] )
    ((starts-with? remark '(you love)) […] )
    ((starts-with? remark '(you hate)) […] )

    (else _____(se '(all you ever say is) remark)_____
      )
    ) )
```

You received 1 pt for each blank filled in correctly. We took off _ a point if it was correct, but had something minor wrong with it (such as an extra set of parentheses). If a recursive solution was attempted (due to the typo on the midterm), then we gave full credit if it was anything reasonable. Including *[…]* in your answer was not correct!

Part B:
Write the function `starts-with?` so that the above code works correctly.

One solution (non-recursive):

```
(define (starts-with? remark first-two)
   (and (equal? (first remark) (first first-two))
        (equal? (first (bf remark)) (first (bf first-two)))
        ))
```

You received 1 pt each for the following:
- Pulling out the first two words of the remark (for the purpose of some kind of comparison).
- Actually doing a correct comparison involving the remark and the other two word sentence given as an argument. Typically to get this point, you needed to actually have `starts-with?` defined to take two arguments.
- Putting these two pieces together correctly and returning a boolean value

_ pt was taken off for minor infractions, such as having extra parentheses

Problem 5 (5 points, 10 minutes): Count-vowels

Write a recursive procedure named `count-vowels` which takes a single word as input, and returns the number of vowels in the word.

Assume that the function `vowel?` has been defined. You may define and use any other helper functions that make sense to you.

For instance,

`(count-vowels 'fred)`	→ 1
`(count-vowels 'frederick)`	→ 3
`(count-vowels 'happy)`	→ 1

This question seemed reasonable for most of you: a recursion down a word in which the recursive case conditionally added 1 to the recursive call, and the base case was a number.

A solution with ifs:

```
(define (count-vowels wd)
  (if (empty? wd)
      0
      (if (vowel? (first wd))
          (+ 1 (count-vowels (bf wd)))
          (count-vowels (bf wd)))
      ))
```

The correct base case was worth 1 point total: 0.5 for getting the test right, 0.5 for getting the return value right. Handling each side of the condition in the recursive case (i.e., when `(vowel? wd)` returned true or false) was worth 2 points.