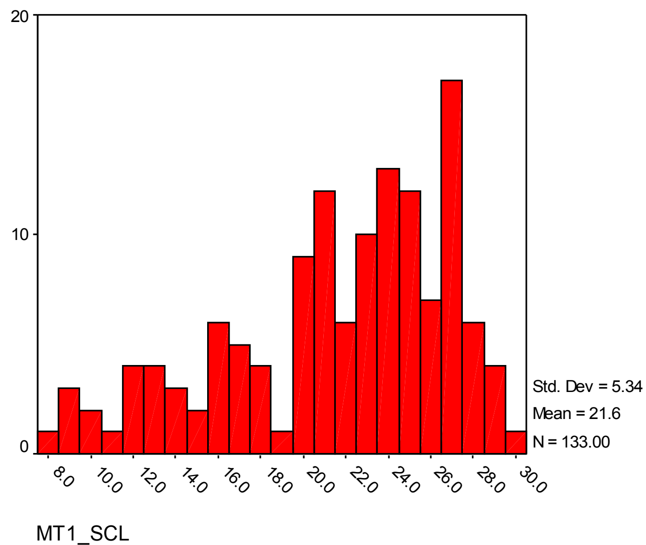
CS3 Midterm 1 **Fall 2006**

Standards and Solutions

Overall, you did good job on this exam.



(9 points). Fill in the blanks.

Write the result of evaluating the Scheme expression that comes before the ➔. If the Scheme expression will result in an error, write *ERROR* in the blank and describe the error.

1	`(count (word 'cs3 'roxor))` ➔ `8`
2	`(count (sentence 'cs3 'roxor))` ➔ `2`
3	`(member? 'dem '(texas democratic party))` ➔ `#f`
4	`(item 3 'a 'b 'c 'd)` ➔ `ERROR - wrong # arguments`
5	`(appearances 'a '(alabama))` ➔ `0` When the second argument is a sentence, appearances checks for words that match the first argument!
6	`(equal? (or 'how 'about 'this)` `(and 'how 'about 'this))` ➔ `#f`
7	`(equal? 'fred (or 'joe 'fred))` ➔ `#f`

8	`(or 'false`
	`(equal? (quotient 7 7) 0))` ➔ `false`
	Remember, the word "`false`" isn't the same as `#f`.

9	`(define (mystery wd)`
	`(if (empty? wd)`
	`""`
	`(word (first wd) 'cs3 (mystery (bf wd)))))`
	`(mystery 'cs3)` ➔ `ccs3scs33cs3`

Those scheming doctors... (A: 6 points, B: 4 points)
Consider a predicate `stressed?` that could be used by doctors to determine whether a patient has encountered a jump in heart rate within a certain time period. `stressed?` takes a sentence of numbers representing heart rates taken each minute, and returns `#t` if two adjacent rates differ by more than 10 beats per minute, and `#f` otherwise.

`(stressed? '(70 71 72 73 74))`	→	`#f`
`(stressed? '(70 71 72 83 84))`	→	`#t`
`(stressed? '(70 71 72 60 59))`	→	`#t`

Part A. Below is a buggy version of `stressed?` with the conditional cases numbered:

	`(define (stressed? rates)`
1	`(cond ((empty? rates) #t)`
2	`((empty? (bf rates)) #f)`

3	`((and (<= (- (first rates) (first (bf rates))) 10)`
	`(<= (- (first (bf rates)) (first rates)) 10))`
	`(stressed? (bf (bf rates))))`

4	`(else #f)))`

For each of the conditional cases above, fill in the corresponding row on the table below:

	Base or recursive case?	**Write OK if it is correct, otherwise write an argument (i.e., a sentence of rates) that will either return an incorrect answer or cause an ERROR *because of this condition*.**
1	Base	Incorrect return value. Any sent with an even number of rates without any jumps was accepted here: e.g., `'()`, `(70 71)`, ...
2	Base	OK We also accepted any sentence of odd length which only contained a jump at the end: e.g., `'(70 71 85)`. It seems better to consider this an error of case #3, however.
3	Recursive	Incorrect recursive call. This case has a fine test, but should recurse only on the `bf` of the rates, rather than the `bf` of the `bf`. With two `bf`s, `stressed?` will miss some of the rate jumps: e.g., `'(70 71 84 85 86)`
4	Base!	Incorrect return value. Any simple jump would show this, e.g., `'(70 85)`

Some common mistakes included:

1. For the right cell of case #3, some of you gave an answer that fit in the associated box for case #4 (e.g., `'(70 85)`), and argued that this should return `#t` when it returns `#f`. This is, in fact, not an error with line 3, as it *should* go to the else case - this is an error in the else case having the wrong return value.
2. A number of you got confused about the difference being being greater than 10. The <= and the `and` were both correct.

The cells in the base/recursive column for the first three cases were worth 1 point if you got all three right. The other five cells in the table were worth 1 point each. Some people listed multiple answers: if you listed at least one correct and one incorrect statement per box, you were given only half a point.

If you wrote out what the error was and a description of a call similar to giving an example you were given the credit - but the question simply asked for the *argument* in a call to the procedure. If you instead wrote the corrected code, this is not what the problem was asking and you were only given one point for the effort.

Part B. A fellow student wants to simplify the 3rd condition by writing a predicate called `within-10?`. This procedure takes two numbers, and returns true if there is a difference of 10 or less between the two numbers.

This student, although a very nice person, isn't the most careful scheme programmer. Write test cases for `within-10?` to fully test the procedure. Write enough test cases to catch all possible bugs, but don't write *too* many: i.e., don't write several test cases that test the same thing.

Make sure you include both the call to `within-10?` as well as the correct return value.

A good solution to `within-10?` might look like this:

```
(define (within-10? x y)
  (and (<= (- x y) 10)
       (<= (- y x) 10)))
```

However, any code that you supplied was ignored: we were only concerned with the test cases. You *did* need to have a general idea of what it would look like, however, to write good tests. Notice that there is no need to test for equality between the two input numbers, and as such, it doesn't make much sense to provide test cases where the numbers are equal. The main focus should be with differences that are around 10, since that is the crucial test. A good strategy would be to test differences of less than 10, 10 exactly, and greater than 10.

Another thing to notice is that you will need to check both when the first number is larger and when the second number is lager. You should do that for each of the various differences. So, something like the following:

```
(within-10? 5 9)   ; #t
(within-10? 9 5)   ; #t

(within-10? 5 19)   ; #f
(within-10? 19 5)   ; #f

(within-10? 5 15)   ; #t
(within-10? 15 5)   ; #t
```

You got 1 point each for a test case with values with a difference less than 10, exactly 10, and greater than 10 respectively. 1 point was also awarded a test cases demonstrating that it didn't matter which of the two inputs was the greater one, although it is certainly a better answer to do this for each of the above differences.

Some common mistakes:

1. Some of you thought `within-10?` took a sentence of two numbers – ie (within-10? '(10 13)) - this isn't what the procedure was supposed to take but we didn't dock since it did have two numbers. Be careful with these issues in your future programming courses, however.
2. Others passed the entire sentence to it like one to stressed, i.e. `(within-10? '(10 13 24 23 14)).` Tis makes no sense in the context of the problem and becomes difficult to grade; at least one point was lost as a penalty. If we could tell which cases you were testing with the sentences we gave you the other possible points.

1. Leaping from Tuesday to Tuesday... (10 points)

Write a function named `tuesday-span` that, when given two dates in 2002, returns the number of Tuesdays in the period spanned by the two dates. January 1, 2002 was a Tuesday (as was December 31). Some examples appear below.

```
(tuesday-span '(january 1) '(january 3))      ➔  1
(tuesday-span '(january 2) '(january 5))      ➔  0
(tuesday-span '(january 6) '(january 9))      ➔  1
(tuesday-span '(january 1) '(january 8))      ➔  2
(tuesday-span '(january 1) '(december 31))    ➔  53
```

You may assume that the first argument date is the same as or earlier than the second argument date. You may use any function appearing in the "Difference Between Dates" code, which works as well for dates in the 2002, without defining it (the "complete" version is included in Appendix A of this test). There are *both* recursive and non-recursive solutions to this problem.

This was a somewhat difficult problem to get exactly right, and as such we were generous in how we awarded partial credit. Your work with day-span and with century-day-span should have been quite a help here: the best non-recursive solutions used a single reference date to get the number of tuesdays – ie., tuesdays since January 1st – and then took the difference for those between the two dates given to `tuesday-span`. To figure out how many tuesdays were before a certain date, you needed to use a procedure to count the number of days (`date-of-year` is perfect, although `day-span` also could work) and then determine how many times 7 could divide into that number (with `quotient`, say).

Just doing the above correctly, which fit in three lines of code, got you 6 or 7 points. However, taking the difference between these two counts won't give the exact right answer because it doesn't take into account when one of the edges fell on a Tuesday. There were two ways to correct for this: the first was more common, and involved writing and using a predicate tuesday? to decide whether to add 1 or not. The second solution involved modifying one of the counts of tuesdays before the reference date by simply checking after the date.

Below is a non-recursive solution using the predicate `tuesday?`:

```
(define (tuesday-span early late)
   (+ (- (tuesdays-since-jan1 late)     ; number of tuesdays before later
         (tuesdays-since-jan1 early))   ; minus the number before the early

      (if (tuesday? Early) 1 0) ))      ; add 1 if this starts on a tuesday
```

```
(define (tuesdays-since-jan1 date)
    (+ 1                ; this correction doesn't matter when the return value
                        ; is subtracted within tuesday-span, but does make the
                        ; answer exactly correct
       (quotient (- (day-of-year date) 1)  ; by subracting one, we correct
                                           ; for the fact that the first
                                           ; tuesday is on the 8th not the 7th
                 7)))

(define (tuesday? date)
    (equal? 0 (remainder (- (day-of-year date) 1) 7)))
```

The other non-recursive solution was something like:

```
(define (tuesday-span early late)
  (-
    (+ 1 (quotient (- (day-of-year late) 1) 7))   ; num tuesdays to later date
                                                  ; inclusive
    (quotient (+ (day-of-year early) 5) 7)        ; num tuesdays before earlier
    ))
```

There were a few recursive solutions, although they all involved the same basic strategy: start at the earlier date and recurse up to the later date, adding 1 for every Tuesday you see. Some of you had trouble realizing you couldn't actually add to a date (e.g., `'(february 14)`), but rather had to convert to a number (like `day-of-year` returns). The easiest way was with a helper function:

```
(define (tuesday-span early late)
  (tuesday-span-helper (day-of-year early) (day-of-year late)))

(define (tuesday-span-helper current-day end-day)
  (cond ((> current-day end-day) 0)                          ; base case
        ((tuesday? Current-day)                              ; rec. case 1
         (+ 1 (tuesday-span-helper (+ 1 current-day) end-day)))
        (else                                                ; rec. case 2
         (tuesday-span-helper (+ 1 current-day) end-day)) ))

(define (tuesday? day)
   (equal? 0 (remainder (- day 1) 7)))
```

The most common mistakes included forgetting to correct for whether the boundary cases fell on a Tuesday (this was worth 3-4 points); using a single call to `day-span` with the two dates, instead of taking the difference of two calls against a reference date (of January 1st, say). In general, you received 2 points for using `day-of-year` (or `day-span`) correctly, 2 points for taking a difference of two calls, and 2 points using `quotient` (or `truncate`) with 7 correctly.

2. Put on your translation hat... (9 points)

Write a procedure `al2en` to translate sentences from "Algolian" to English. Algolian sentences have the following rules, which differ from English:

- Sentences are either questions or statements. In Algolian questions, a "?" will be the first element of the sentence. In English questions, the "?" should go at the end. Statements have no punctuation.
- In an Algolian statement, the subject is in the second position. In English, the subject is in the first position, followed by the verb.

- In an Algolian question, the subject is in the third position. In an English question, the verb is in first position followed by the subject.
- In Algolian, verbs are always at the end of the sentence, and always have an extra "`ing`" at the end of the word. In English, verbs don't have the extra "`ing`", and they go first if the sentence is a question, or second if the sentence is a statement.
- The first word in Algolian (that isn't a "`?`") can't be translated into English. Simply remove it from the corresponding English sentence. All other words not covered by one of these rules should be copied verbatim.

Examples will make this easier to understand.

`(al2en '(doying I friend with joe bob aming))`	→	`(I am friend with joe bob)`
`(al2en '(? boing you to dinner cominging))`	→	`(coming you to dinner ?)`
`(al2en '(? pwing joe fish eating))`	→	`(eat joe fish ?)`

The resulting sentences aren't necessarily valid English, as the examples show. Use helper functions in order to make your code easier to read *and* write.

This question seemed to be very easy for most of you, with the majority of you getting all 9 points. The solution involved straightforward manipulation of sentences and words. We emphasized the use of helper functions only partially because they made your code soooo much easier to read! One solution:

```
(define (question? al-sent)
  (equal? (first al-sent) "?"))

(define (verb al-sent)
  (bl (bl (bl (last al-sent)))))

(define (subject al-sent)
  (if (question? al-sent)
      (first (bf (bf al-sent)))
      (first (bf al-sent))))

(define (the-rest al-sent)
  (if (question? al-sent)
      (bl (bf (bf (bf al-sent))))
      (bl (bf (bf al-sent)))))

(define (al2en al-sent)
  (if (question? al-sent)
      (se (verb al-sent) (subject al-sent) (the-rest al-sent) "?")
      (se (subject al-sent) (verb al-sent) (the-rest al-sent))
      ))
```

This problem was graded based on the following standard:

Few or no helper functions: -1

If there were major problems with any of the following: -1

- the `verb` procedure
- the `subject` procedure

- the `the-rest` procedure (names varied here)
- appending "?" to the question sentence
- removing the untranslateable word

Problems testing whether a question or a statement was given: -1
Other syntax errors (e.g., inappropriate use of `butfirst` and `first`): -1

Some common mistakes included not using a helper function to solve the problem; and confusion with how many `butfirst` calls to make to retrieve the appropriate parts of the input sentence.

3. Good help is hard to find... (A: 10 points, B: 4 points)

This question concerns a database of *tutors*, and the use of good abstraction. A tutor entry is stored in scheme as a word, with 1-3 parts separated by asterisks (*).

- The first part is the tutor's <u>name</u>, and must be present. It can be any length, and will not contain an asterisk.
- The second part is the tutor's discipline, and may or may not be present. Disciplines are always non-numeric, and may be any length, and will not contain an asterisk. You won't have to work with this part, other than recognizing that it could be in a tutor entry.
- The third part is the tutor's <u>rating</u>, and may or may not be present. This is an integer between 1 and 6. If the rating is missing from the entry, that tutor is considered to have a rating of 0.

The following are all valid entries for tutors:

```
Andre_3000
joe*English
Peter_Jennings*2
Sade*Music*5
```

Part A. Write two accessors for tutor entries: one for the <u>name</u> and one for the <u>rating</u>. (Do not write one for discipline). Both procedures take a tutor entry as the single parameter. The accessor for name will return the name as a word; the accessor for rating will return an integer between 0 and 6. Be sure to use meaningful names for your procedures and arguments.

For both of these accessors you need to pull apart a word into smaller words. The `name` accessor required recursion because you the end of the name could be any length (and, the end of the name was not always a fixed length from the end of the word). Essentially, you needed to recurse down the tutor entry word, building up a word as you went along, and stopping when you hit an asterisk *or* the end of the word. That makes 2 base cases, and one recursive case:

```
(define (name tutor)
  (cond ((empty? tutor) "")
        ((equal? (first tutor) '*) "")
        (else (word (first tutor)
                    (name (bf tutor))))
        ))
```

For the rating accessor, you needed to test to see whether the tutor entry had a rating, which involved two tests. First, that the last element of the word was a number, to make sure that the tutor entry didn't end in a discipline. And second, that the second to last element was an asterisk, to make sure that the entry didn't only contain a name. Just checking for one or the other didn't rule out all the cases where the rating wasn't there (i.e., an entry with just a name ending in a number, or a entry with a one-letter discipline).

```
(define (rating  tutor)
  (if (and (number? (last tutor))
           (equal? '* (last (bl tutor))))
      (last tutor)
      0))
```

The `name` procedure was worth 6 points, `rating` was worth 3 points. For small errors, you lost 1 point; for larger ones, 2 points.

There were a variety of common errors:

1. Trying to use `'()` in a base case for `name`
2. Forgetting the empty base case when looking for the name (this matters for when you're given only a name for a given tutor)
3. Forgetting to quote the asterisk (i.e., `*` instead of `'*`).
4. Only returning the last of the input in the rating function, irregardless if a rating actually exists (this usually resulted in a 2 point loss).
5. Returning the last of the input if they find a `*` in the input to rating, but not checking if the `*` seperated the name and the tutor's discipline.

Part B. Write a predicate `hire?`, which takes a tutor entry and a sentence of "bad" tutor names. `hire?` Should return `#f` if the tutor should not be hired, and `#t` otherwise. Tutors should not be hired if their ratings are below 4 or if their names appear within the sentence of bad tutor names.

`(hire? 'Sade*music*5 '(peter_jennings andre_3000 joe))`	→	`#t`
`(hire? 'Sade*music*5 '(peter_jennings andre_3000 joe Sade))`	→	`#f`
`(hire? 'joe '())`	→	`#f`

Use proper abstraction.

The key to getting this problem write was to use the accessors that you had written in Part A:

```
(define (hire? tutor bad-name-list)
  (and (> (rating tutor) 3)
       (not (member? (name tutor) bad-name-list))))
```

Grading here was similar to that of 5a: -1 was taken for most errors Forgetting to use the name and rating accessors you wrote resulted in -2 points.

Most errors were slight logic problems:

1. Checking if rating was "`> 4`" for a bad tutor, which excluded tutors with rating 4.
2. Conversely, checking if rating was "`< 4`" for a good tutor
3. Thinking that an empty bad tutors list meant that the input tutor was automatically bad.