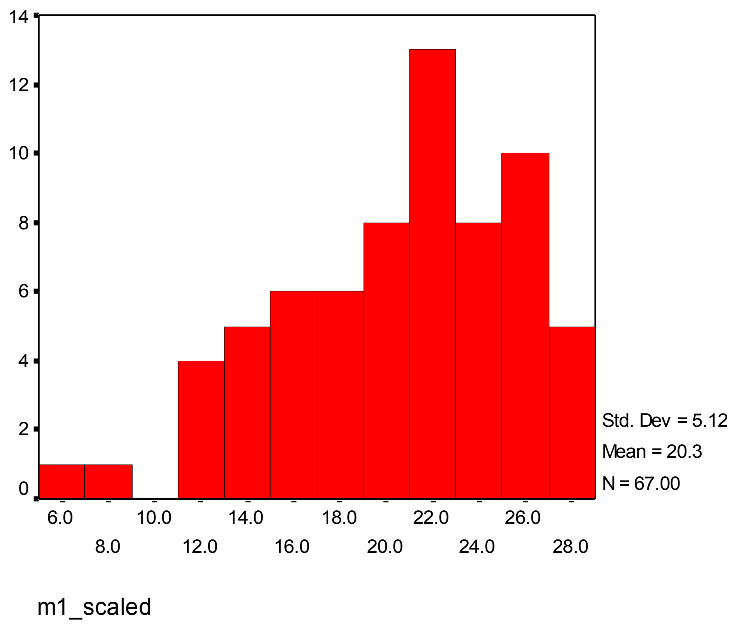
CS 3 Midterm 1
Spring 06
Standards and Solutions

The overall histogram of scaled scores (out of 30) looks like:



Problem 1 (8 points). Fill in the blanks.

	`(first (last (first (last 'california))))` ➔ `a`
	`(appearances 'john 'john)` ➔ `0` The second argument is a word, not a sentence; hence, the answer is 0.
	`(equal? 'x (or 'y 'x))` ➔ `#f` This one tricked a lot of you, because of your commonsense notion of `or`. The scheme interpreter, though, does the `or` *first*, and evaluates it to `y`. Clearly, then, the `equal?` statement is false.
	`(equal? '(word) ('word))` ➔ `ERROR: 'word isn't a procedure`

	`(or 'false`
	`    (= 3 (+ 2 1)))` ➔ `false`
	
	Remember, `or` returns the first true thing it finds or `#f`. In this case, the first true thing is the word `false`.

(6)	`(and (quote a) (quote b) (quote quote))` → `quote` `and` returns the last true thing it finds, or `#f`. The word `quote` (which was quoted in the statement) evaluates to true.
(7)	`(day-span '(december 32) '(december 32))` → `1`

(8)	`(day-span (day-span '(january 1) '(january 3))`
	`(day-span '(december 29) '(december 31)))` →
	
	`ERROR: the outer day-span is not given dates as arguments`

These were each worth 1 point, and you pretty much had to have it completely right to get that point. Some common problems were on (2), with `appearances`, and (5) and (6), where many of you seem to think that `and` and `or` can only return `#t` or `#f`, rather than the value that evaluates to `#t`.

Problem 2 (A: 4 points, B: 12 points) Roman numerals.

This question concerns the "Roman Numerals" case study. The "rewritten" code is reproduced at the end of this exam, in appendix A.

Part A. What will `decimal-value` return for these calls? (These may not be legal roman numerals). If an error results, write ERROR and say which procedure will cause the error.

`(decimal-value 'vvviiii)` →	`19`
`(decimal-value 'ximcm)` →	`1909` Remember, `im` is a prefix here.
`(decimal-value 'abcde)` →	`ERROR, in decimal-digit-value or sum-all`
`(decimal-value '(m c m x c i x))` →	`1999`

These were each worth a point. For the third one, you lost half a point if you didn't say where the error occurred; we accepted either `decimal-digit-value` or `sum-all`.

Part B. The rewritten procedure `prefix-values-removed` consists of four `cond` clauses, identified below:

```
    (define (prefix-values-removed number-sent)
      (cond
1       ((empty? number-sent) '( ))
2       ((empty? (bf number-sent)) number-sent)
3       ((not (starts-with-prefix? number-sent))
         (se (first number-sent) (prefix-values-removed (bf number-sent))))
4       ((starts-with-prefix? number-sent)
         (se
           (- (first (bf number-sent)) (first number-sent))
           (prefix-values-removed (bf (bf number-sent))) ) ) ) )
```

Consider what will happen if one of these `cond` clauses is missing completely. Fill in the table below. When asked for an argument, write a value for `number-sent` that will cause the indicated outcome. Use only arguments that *might normally come up in the operation of* `decimal-value` on proper roman numerals.

> Note, you had points taken off if you provided arguments that couldn't occur. For instance, some of you provided (5 10) as an argument, but that can't be derived from a legal roman numeral!

Missing Clause	**Is this a Base or Recursive case? Circle one.**	**Argument that will cause an ERROR or an incorrect value**	**Argument that will return a correct result**
2	[✓ Base] Recursive	`(1000 1000 10)` Anything that doesn't end in a prefix	`(1000 1000 1 10)` Anything that ends in a prefix
3	Base [✓ Recursive]	`(100 1000 50)` Anything that contains a non-prefix	`(100 1000 1 5)` Anything that contains only prefixes
4	Base [✓ Recursive]	`(100 1000 50)` Anything that contains a prefix	`(1000 50 1)` Anything that contains only non-prefixes

> The top two boxes were worth 2 points each; the bottom four boxes were worth 6 total.
>
> The majority of you did quite well on this problem, which is great: we weren't sure if this was going to be difficult for the class. The most common error was swapping two of the cells horizontally. For instance, thinking that leaving out clause 4 would result in an error when an argument contains only non-prefixes.

Problem 3 (A: 3 points, B: 3 points, C: 4 points) What comes between?

Write a procedure called `between?` which takes three numbers as arguments, and returns true if and only if the second argument is between and not equal to the first and the third:

```
(between? 5 6 7) → #t
(between? 7 6 5) → #t
```

Part A: Write `between?` without using `if` or `cond`.

> Answers were all over the map here! A common solution, and one we like best:
>
> ```
> (define (between? bound1 mid bound2)
>     (or (and (< bound1 mid) (> bound2 mid))
>         (and (< bound2 mid) (> bound1 mid))))
> ```

A lot of you did not notice that both `(between? 5 6 7)` and `(between? 7 6 5)` returned true! 2 points were deducted if you only handled one of these possibilities. 1 point was taken off if you had some issues with equal numbers.

Part B: Write `between?` without using `and` or `or`.

```
(define (between? bound1 mid bound2)
    (cond ((< bound1 mid) (> bound2 mid))
          ((< bound2 mid) (> bound1 mid))
          (else #f)))
```

This part has the same grading standards as *Part A*. 1 point was taken off if the `else` case was not included, or, basically, not handling the case of equal numbers. 1 point was taken off if the code was overly complicated and unnecessarily long. You received either zero points or 0.5 points if you used `and` or `or` in this part when you were not allowed to.

A note on coding style (no points taken off):
A number of you wrote your `cond` clauses like this:

```
    (cond ((< bound1 mid) (if (> bound2 mid) #t #f)) …)
```

The use of `if` is unnecessary and redundant here. Instead, the clause could have been shortened to `((< bound1 mid) (> bound2 mid))`

Consider this: `(define (bar x y) (if (< x y) #t #f))`. The use of `if` is repetitive because it basically says to return true if `(< x y)` evaluates to true, and return false, if `(< x y)` evaluates to false. A better way to define `bar` would be `(define (bar x y) (< x y))`, so `bar` returns directly whatever the expression `(< x y)` evaluates to. Though both versions return the same results, style should certainly be a part of your consideration in coding.

Part C: Write a suite of test cases for `between?`. Make sure you test the possible sets of parameters exhaustively as possible, in order to test different ways the code could be written.

Also, make sure you describe what the result of the call should be!

The set of test cases is meant to test other people's code, not your own. This means that you cannot assume anything about how they coded this procedure. Thus the test cases must be as exhaustive as possible to handle the different ways other students could have coded this same problem.

One point was given to the two cases that can make `between?` return true.

```
    (between? 5 6 7) → #t
    (between? 7 6 5) → #t
```

Two points were assigned to the handling of out of bounds cases. This means that `mid` is outside the range of `bound1` to `bound2`, where `bound1` and `bound2` are not equal to each other. In order to receive two points, you needed a test case with a `mid` that is greater than both `bound1` and `bound2`, and another test case with a `mid` that is less than both `bound1` and `bound2`.

```
(between? 5 9 7) → #f
(between? 7 9 5) → #f
(between? 5 1 7) → #f
(between? 7 1 5) → #f
```

One point was allocated to the following set of test cases. This group of tests check for equal numbers among the three arguments. You needed to check for at least two of these possibilities.

```
(between? 7 6 6), (between? 6 6 7) → #f
(between? 7 7 6), (between? 6 7 7) → #f
(between? 5 8 5), (between? 5 3 5) → #f
(between? 2 2 2) → #f
```

Problem 4 (13 points) A better interpreter.

Recall the procedure `interpret` that you wrote in a quiz during lab. This procedure took a sentence as input and returned what Scheme would have returned had you typed it in. For instance,

```
(interpret '(+ 3 7))  ➔  10
```

This question involves a more fully functioning interpreter (you won't write the complete procedure):

1. More possible procedures: For this problem, the interpreter understands input sentences that start with the word

   `+`, `-`, `sqrt`, `between?`, `word`, or `sentence`.

   (`between?` refers to the procedure discussed in question 3)
2. Input sentences can be longer: The input sentence will have at least one word in it, indicating the procedure. It may have any number of additional words in it. As with all sentences, the input sentence will contain only words.
3. Error checking: For some kinds of input sentences that don't make sense, the interpreter will return a sentence containing an relevant error message. The interpreter checks
   a. Whether the first word names one of the procedures it can handle
   b. Whether the number of arguments is correct
   c. Whether the type of arguments is correct

Your job is to write the procedure `invalid-scheme?`, which would be used as a helper procedure for the better interpret procedure. It will check whether the input sentence is valid; it does *not* calculate the actual result. `Invalid-scheme?` returns:

- `#f`, if the sentence fulfills the requirements above (i.e., is valid), and can be interpreted.
- The error sentence `(expected <number> parameters)`, where `<number>` is an integer
- The error sentence `(expected only numbers)`
- The error sentence `(unknown procedure)`

On the following page is an incomplete version of `invalid-scheme?`. Fill in the blanks and boxes so that it works correctly.

```
;; Checks whether a sentence is invalid scheme, as detailed earlier
(define (invalid-scheme? sent)
   (cond ((and (member? (first sent) '(+ - sqrt between?))
               (all-numbers? (bf sent)))
          (cond ((or (equal? (first sent) '+)
                     (equal? (first sent) '-))
                 #f)

                ((and (equal? (first sent) ___'sqrt___)

                      (not (equal? (count sent) ____2_____)))
                 '(expected 1 parameter))

                ((and (equal? (first sent) _____'between?____)

                      (not (equal? (count sent) _____4_________)))
                 '(expected 3 parameters))

              __(else #f)____________________________________________
                ))
          ((member? (first sent) '(+ - sqrt between?))

            _'(expected only numbers)______________________________
            )

           ((member? (first sent) '(word sentence))
            #f)

           (else '(unknown procedure))

      ))    ;;end invalid-scheme?
```

```
;; predicate that indicates whether a sentence is all numbers
(define (all-numbers? sent)
   (if (empty? sent)
      #t
      (and (number? (first sent))
           (all-numbers? (bf sent)))))
```

Some common errors include:

Using 1 and 3 rather than 2 and 4 in the second and fourth boxes: this lost 1 point each.

Not quoting `sqrt` or `between`: minus 1 point for each.

The `(else #f)` had many strange answers in it, like `#t` or some error message. This blank was worth 1 point.

The boxes were worth 3 and 4 points respectively. The most common mistake for the first box was to leave it blank. You mostly did quite well on the last box (`all-numbers?`). Several of you did something like `(member? (first sent) '(0 1 2 3 4 5 6 7 8 9)` rather than `number?`, which lost one point.

Problem 5 (12 points) Sub-cursion?

Write the procedure `sub-sentence`, which returns a middle section of a sentence. It takes three parameters; the first identifies the index to start the middle section, and will be 1 or greater; the second identifies the length of the middle section, and will be 0 or greater; and the last is the sentence to work with.

Do *not* use any helper procedures.
Do *not* use the `item` procedure in your solution.

```
(sub-sentence 2 3 '(a b c d e f g)) ➔ (b c d)
(sub-sentence 3 2 '(a b)) ➔ ()
(sub-sentence 3 0 '(a b c d e) ➔ ()
(sub-sentence 3 9 '(a b c d e) ➔ (c d e)   errata
```

This is an accumulating recursion: in some fashion, you need to count down *two* different times: once to get to the starting position and another time to get to the end of the sub-sentence (i.e., counting down length). For the later recursive call, you will need to be collecting up the sentence.

```
(define (sub-sentence start len sent)
   (cond ((empty? sent)
          ;; stop the recursion if our sentence ends,
          ;;  no matter what has happened yet
          '())
         ((> start 1)
          ;; we need to get to the starting position
          (sub-sentence (- start 1) len (bf sent))
         ((> len 0)
          ;; we've made it to the starting position, and now we
          ;; need to gather up the sub-sentence while moving
          ;; to the end of the sentence
          (se (first sent)
              (sub-sentence start (- len 1) (bf sent))))
         (else
          ;; we're at the end of the sub-sentence (len=0), so
          ;; we can just stop now.
          '())
```

```
        ))
```

Most of you put the last base case (where `len` is `0`) at the top, but notice that it works fine after the recursive cases! The else clause also catches the situation where the start argument is greater then the `count` of `sent`. It also catches the situation where the `(+ len start)` goes beyond the `count` of `sent`.

A very few of you changed the second recursive call to lose words off the end of the sentence, and changed the base case to return the current sentence when the length of that sentence was equal to (or less than) the length of the sentence you were to return. This was a neat solution.

Some common errors include:

Bad placeholder names

Forgetting to check for an empty list

Trying to remove elements from the end of the list (it is possible, but it is more difficult