172 students took the exam. Scores ranged from 2 to 40. The mean was 29.4; the median was 31. 93 students scored between 30.5 and 40, 54 between 20.5 and 30, 22 between 10.5 and 20, and 3 less than 10.5. Were you to receive grades of 75% on each in-class exam (*i.e.* 30 on this exam) and on the final exam, plus good grades on homework and lab, you would receive an A–; similarly, a test grade of 20 may be projected to a B–.

There were two versions of the test. (The version indicator appears at the bottom of the first page.)

If you think we made a mistake in grading your exam, describe the mistake in writing and hand the description with the exam to your lab t.a. or to Mike Clancy. We will regrade your entire exam.

Problem 0 (2 points)

Each of the following cost you 1 point on this problem: you earned some credit on a problem and did not put your login name on the page, you did not adequately identify your lab section, or you failed to put the names of your neighbors on the exam. The reason for this apparent harshness is that exams can get misplaced or come unstapled, and we want to make sure that every page is identifiable. We also need to know where you will expect to get your exam returned. Finally, we occasionally need to verify where students were sitting in the classroom while the exam was being administered.

Problem 1 (4 points)

This problem requested that you add parentheses and quotes to a call to `cons` and a call to `append` to produce the list `((A B) C)` on version A or the list `((F G) H)` on version B. Here are solutions for version A:

```
(cons '(A B) '(C))

(append '((A B)) '(C))
```

The calls were worth 2 points each. Deductions are listed below. If a particular error was made in each part, points were deducted in each part except as noted.

–1	one misparenthesized argument
–2	two misparenthesized arguments
–2	using the wrong number of arguments to `cons`
–1	any misquoting
–.5	omitting one of the two quotes
–.5	omitting the outside parentheses (deducted only once)

Problem 2 (6 points)

This problem was the same in both versions.

Part a asked you where a crash occurs when `ttt` is called with a full board. With such a call, all of `i-can-win?`, `opponent-can-win?`, `i-can-fork?`, and `i-can-advance?` return `#f`. Finally `best-free-square` is called, which calls `first-choice`, which calls `first` on an empty sentence.

This part was worth 2 points, 1 for the procedure and 1 for the explanation. Generally you had to mention `first-choice` to earn any points.

Part b, worth 4 points, was to detect a full board represented as a sentence of triples. The most popular correct answer was

```
(define (all-squares-filled? triples)
  (empty? (keep integer? (accumulate word triples)))
```

An alternative was to handle each triple separately:

```
(define (all-squares-filled? triples)
  (empty? (keep has-blank-space? triples))

(define (has-blank-space? triple)
  (not (empty? (keep integer? triple))))

(define (has-blank-space? triple)
  (< (+ (appearances 'o triple) (appearances 'x triple)) 3) )
```

By far the most common error was to compare an integer or a square to a triple, for instance by saying

```
(keep integer? triples)
```

A similar error was to compare an integer to a word of triples:

```
(number? (accumulate word triples))
```

Both these errors lost 2 points.

Another common error was to assume the argument was in position format, i.e. a word containing `x`'s, `o`'s, and `_`'s, rather than a sentence of triples. This simplified the problem significantly,

```
(define (all-squares-filled? position)
  (not (member? '_ position)) )
```

and thus lost all 4 points for this part.

Some other deductions are listed below. No points were deducted for parenthesis errors.

–1	a small logic error, e.g. switching true for `#f`
–1.5	two small logic errors
–1	building a sentence of empty words, then checking if it's empty
–2	using `every` instead of `keep`
–4	using `possibilities` or `preferences` in an attempt to copy `first-choice`

Problem 3 (5 points)

Part a of this problem was to supply a good comment for the procedure

```
(define (mystery sent)
  (= 1 (remainder (accumulate + sent) 2)) )
```

(In version B, the comparison was with 0 rather than 1.) As was explained in lab, a good comment describes the argument and the result value: `sent` is a sentence containing *only* integers (otherwise the call to + or to `remainder` would crash), and the value returned is true if the sum of the integers is odd (even in version B) and `#f` otherwise.

This part was worth 2 points, 1 for describing `sent` and 1 for describing the return value. To earn the `sent` point, you had to specify that `sent` consists only of integers (we accepted "numbers" too). To earn the result point, you had to use the word "odd" or "even"; merely parroting back the code was insufficient.

In part b, you were to complete the recoding of `mystery`, moving the `remainder` call into the `accumulate`. The answer:

```
(define (mystery2 sent)
  (= 1
    (accumulate
      (lambda (a b) (remainder (+ a b) 2))
     sent) ) )
```

This part was worth 3 points. You needed a two-argument procedure and a `lambda` body to earn any points on this part. 1 point was deducted for each flaw in the `lambda` body, for example:

arithmetic syntax for `remainder` instead of Scheme syntax;

use of entire sentence for one of `accumulate`'s arguments instead of individual elements;

wrong computation (e.g. `(+ (remainder a 2) (remainder b 2))`)

Problem 4 (6 points)

This problem was the same in both versions. It asked you to find three illegal Roman numerals for which `decimal-value` returns 5, and to list calls to `roman-sum` that are made as a result of each computation. Here are four such numerals.

illegal Roman numeral	*resulting calls to* `roman-sum`

`iiiii`	`(roman-sum (1 1 1 1 1))`
	`(roman-sum (1 1 1 1))`
	`(roman-sum (1 1 1))`
	`(roman-sum (1 1))`
	`(roman-sum (1))`

`ivi`	`(roman-sum (1 5 1))`
	`(roman-sum (1))`

`iiv`	`(roman-sum (1 1 5))`
	`(roman-sum (1 5))`
	`(roman-sum ( ))`

`vx`	`(roman-sum (5 10))`
	`(roman-sum ( ))`

Each correct illegal Roman numeral earned you 1 point; each correct set of calls to roman-sum earned you 1 point. An incorrect Roman numeral probably lost both points.

You lost ½ point for each inclusion of a call to roman-sum with an empty argument where it should not have appeared (for iiiii and ivi), and for each omission of that call where it should have appeared (for iiv and vx). You lost 1 point for a call to roman-sum that would not have appeared in a correct trace.

Omitting all the recursive calls to roman-sum lost you 2 or 3 points, depending on how much information you provided about how the result 5 was determined.

Problem 5 (9 points)

In this problem, you completed an accumulating recursion for the split procedure. The problem was the same on both versions. Here is a solution; code you were to add is underlined and numbered.

```
(define (split wd)
   (helper
      wd

      ""          ; 1: OK to say (word) or (word "")

      '( ) ) )    ; 2: OK to say (sentence)
(define (helper wd wd-so-far sent-so-far)
   (cond
      ((empty? wd) (sentence sent-so-far wd-so-far )  ; 3:
      ((equal? (first wd) "|")
       (helper
         (bf wd)

         ""                                   ; 4:

         (sentence sent-so-far wd-so-far) ) )   ; 5:
      (else
       (helper
         (bf wd)

         (word wd-so-far (first wd))          ; 6:

         sent-so-far ) ) ) )                  ; 7:
```

This solution involved nested accumulations, one to build up each word, the other to include successive words into the result sentence.

Points were allocated as follows.

- Blanks #1 and #2: ½ point each. A common answer that received a ½-point deduction was `'("")`; see below for a reason that this deduction might be incorrect.
- Blank #3: 2 points, 1 for `sent-so-far` and 1 for `wd-so-far`. Common answers were `sent-so-far` (1 point deducted, but see below), `(sentence wd-so-far)` (1 point deducted), and `'( )` or `""` (2 points deducted).
- Blank #4: 1 point.
- Blank #5: 2 points. A common error was `(sentence wd-so-far)`, which lost 1 point.
- Blank #6: 2 points. A common error was `(word (first wd))`, which lost 1 point.
- Blank #7: 1 point.

Overall, points were distributed into three categories: 1 for initialization, 4 for accumulation of the characters in each word, 4 for accumulation of the sentence.

Another common error was to switch the arguments in blanks #3, 5, and 6, thereby building each word or the sentence in reverse order. Switching the arguments in blanks #3 and 5 lost ½ point, as did switching the arguments in #6. All parenthesis errors collectively were worth a ½-point deduction. Extraneous empty words also lost ½ point, but see below.

Including an extra call to `split` or `helper` in any of the `cond` cases lost the points for that case (1, 3, or 3 respectively).

After the grading, we became aware of an alternate solution, shown below. The underlying approach is to ignore the third argument to `helper`, and have the second argument keep track of the state of the accumulation. Here is the code; the ignored third argument to `helper` is omitted, and code to be added is underlined.

```
(define (split wd)
  (helper wd '("")) )               ; 1

(define (helper wd so-far)
  (cond
    ((empty? wd) so-far)            ; 2
    ((equal? (first wd) "|")
     (helper (bf wd) (sentence so-far "")) )     ; 3
    (else
     (helper
       (bf wd)
       (sentence                                 ; 4
         (bl so-far)
         (word (last so-far) (first wd)) ) ) ) ) )
```

Points for this approach would be allocated as follows.

- Blank #1: 1 point.
- Blank #2: 1 point.
- Blank #3: 2 points.
- Blank #4: 5 points, 3 for the call to `sentence`, 2 for the call to `word`.

If your solution took this approach, we would be happy to reevaluate it using the point allocations just described.

Problem 6 (8 points)

This problem involved the analysis of code to verify that a given word or sentence is the result of compression as implemented in an earlier homework. Versions differed only in the sequence of parts a-c.

You were asked to identify sample arguments that would be incorrectly classified by the code, either because a true value is returned when `#f` is desired, `#f` is returned for a correct result of compression, or a crash happens.

- Incorrect classification of a correct result of compression is not possible.
- A sentence that's not a correct result of compression for which `result-of-compression?` returns true is `(1 1)`. In general, any single 1 or group of 1's (e.g. `(4 1)`) following another single 1 or a group of 1's produces a true value incorrectly. (The same applies to single 0's or groups of 0's.)
- A sentence that ends with a value greater than 1, which is not the result of compression, results in a crash.

The three examples were worth 1 point each. Common errors were misclassification of an empty sentence (a correct result of compression; we reminded you at the exam that `true-for-all?` should return true for an empty argument) or a sentence containing nonnumbers (ruled out by the first `true-for-all?` call).

Part b involved locating and explaining the bugs. (You didn't need to fix them.) There are three bugs:

- `legal-starting-with-compressed?` in the second `and` clause accesses the second word in its argument sentence without making sure that the sentence contains at least two words.
- Both `legal-starting-with-uncompressed?` and `legal-starting-with-compressed?` fail to check the next item or group to make sure that as much as possible of the sentence is compressed. To identify these bugs, you could have noted either that extra `and` clauses are needed, or that the recursive call is made prematurely.

This part was worth 5 points, 1 each for the three bugs, 1 each for the two explanations. You lost 1 point for each missing item and ½ for each vague explanation. Common errors were identifying non-bugs, identifying a bug without providing an explanation, and giving the same explanation twice.