177 students took the exam. We are still gathering statistics about the exam (mean and median). However, were you to receive grades of 75% on each in-class exam and on the final exam, plus good grades on homework and lab, you would receive an A–; similarly, a test grade of 21 may be projected to a B–.

There were four 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 (6 points)

In this problem, you were to evaluate the results of calls to a procedure named `how-built`:

```
(define (how-built x)
  (sentence 'sentence (first x) (butfirst x)) )
```

Versions differed in the arguments used and the sequence of calls. Version A's answers are as follows.

expression	*result*
`(how-built '(a b c d))`	`(sentence a b c d)`
`(how-built 'abcd)`	`(sentence a bcd)`
`(how-built '(a))`	`(sentence a)`
`(how-built 'a)`	`(sentence a "")`

The first two answers were worth 1 point each. The last two answers were worth 2 points each. Some special cases:

–1 Omit parens or include quotes.

–2 Omit `sentence`.

–1 `(sentence a ( ))` instead of `(sentence a)`. (Sentences can contain only words, not sentences.) The rationale for awarding partial credit is that at least the student knows what `(butfirst '(a))` is.

–1 `(sentence a)` instead of `(sentence a "")`.

Problem 2 (5 points)

This problem involved working with and rewriting a procedure named `exam-proc`. The four versions of `exam-proc` appear below.

```
; Version A

(define (exam-proc x)
  (cond
    ((not (member? x '(1 3 5 7 9))) #T)
    ((member? x '(2 3 4 5)) #F)
    (else #T) ) )
```

```
; Version B

(define (exam-proc x)
  (cond
    ((not (member? x '(1 3 5 7 9))) #T)
    ((member? x '(4 5 6 7)) #F)
    (else #T) ) )
```

```
; Version C

(define (exam-proc x)
  (cond
    ((not (member? x '(0 2 4 6 8))) #T)
    ((member? x '(2 3 4 5)) #F)
    (else #T) ) )
```

```
; Version D

(define (exam-proc x)
  (cond
    ((not (member? x '(0 2 4 6 8))) #T)
    ((member? x '(4 5 6 7)) #F)
    (else #T) ) )
```

In part a, you were to evaluate a call to `exam-proc`:

version	*call*	*result*
A	`(exam-proc 3)`	`#F`
B	`(exam=proc 9)`	`#T`
C	`(exam-proc 2)`	`#F`
D	`(exam-proc 8)`	`#T`

This was worth 1 point; no partial credit was awarded.

Part b involved converting the `cond` to a call to `member?`. To do this, one might test each of the ten values separately. A more efficient approach, however, might go like this (using version A):

- 0, 2, 4, 6, 8 all produce a true value.
- 3 and 5 produce a false value (2 and 4 were covered by the preceding case).
- What's left—1, 7, and 9—also return a true result.

Thus the argument to `member?` is `'(0 1 2 4 6 7 8 9)`. Corresponding answers to versions B, C, and D are `'(0 1 2 3 4 6 8 9)`, `'(0 1 3 5 6 7 8 9)`, and `'(0 1 2 3 5 7 8 9)` respectively.

This part was worth 2 points. Common errors and corresponding deductions were as follows.

- You lost 1 point for forgetting about 0.
- Many students thought that some values left out of the sentence in the first `cond` clause—2 and 4 in version A, 4 and 6 in B, 3 and 5 in C, and 5 and 7 in D—don't return true; instead, they thought that execution proceeded to the second clause and returned `#f` for these values. This lost you 2 points.

- A similar misconception was that only values left out of both `member?` calls could return true: 6 and 8 for version A; 2 and 8 for B; 1, 7, and 9 for C; and 1, 3, and 9 for D. This was also a 2-point error.

Finally, you were to rewrite `exam-proc` without using `cond` or `if`, making sure that it handles non-integer and out-of-range integer arguments the same as the original code (*i.e.,* returns true). A version using the builtin `integer?` procedure is

```
(define (exam-proc x)
  (or
   (not (integer? x))
   (< x 0)
   (> x 9)
   (member? x your_answer_to_part_b )) )
```

Use of `integer?` turns out not to be necessary, however:

```
(define (exam-proc x)
  (not (member? x single_digits_not_in_your_part_b_answer)) )
```

The argument to `member?` here is a sentence containing the two single-digit values for which `exam-proc` returns `#F`. In order not to dock you twice for the same part b error, we used your answer to part b to determine the argument to `member?`.

This part was worth 2 points: 1 for correctly handling non-integers and 1 for correctly handling integers outside the range of 0 to 9.

Problem 3 (6 points)

This problem explored the effect of passing *words* as arguments to the version 1 day-span rather than sentences. One might expect that this would always cause a crash, but that's not necessarily the case. The procedures that access date information, `month-name` and `date-in-month`, each returns a character value for a word argument of two or more characters. Some of the procedures in the program manage to avoid crashing with character arguments.

In part a, you analyzed a call that does produce a crash, and were asked to describe where and why the crash occurred. First, `same-month?` is called. It compares the first character of the first argument with the first character of the second argument. These match, so `same-month?` returns true. Then `same-month-span` is called. It subtracts the second character in the first argument from the second character in the second argument. The second argument in all the versions is a letter, which is an illegal argument for the subtraction procedure.

This part was worth 2 points. A common error, earning a 1-point deduction, was to explain correctly how a error would occur in `consecutive-months?`.

For part b, you had to fill in the blank with a word argument for which `day-span` would not crash. Continuing with the call to `same-month-span`, we see that any word whose first character matches that of the given argument and whose second character is a digit works.

You could earn 1 point for the answer and 3 points for the explanation in this part. A 1-point deduction resulted from assuming that `date-in-month` could return a two-digit value.

Problem 4 (9 points)

This problem was the same in all versions. It asked you to consider a calendar in which all the months with 31 days preceded all the months with 30 days, leaving a 28-day December at the end. In part a, you were to implement `equal-length-month-span` for this calendar, using procedures already defined in the version 1 code wherever possible. Here's a solution.

```
(define (equal-length-month-span earlier-date later-date)
  (+
    (days-remaining earlier-date)
    (date-in-month later-date)
    (*
      (days-in-month (month-name earlier-date))
      (-
        (-
          (month-number (month-name later-date))
          (month-number (month-name earlier-date)) )
        1) ) )
```

The calls to `days-remaining` and `date-in-month` can but need not be replaced by a single call to `consec-month-span`. Here's an equivalent version that uses `same-month-span`; note in particular that it handles the case of a negative return value.

```
(define (equal-length-month-span earlier-date later-date)
  (+
    (*
      (days-in-month (month-name earlier-date))
      (-
        (month-number (month-name later-date))
        (month-number (month-name earlier-date)) ) )
    (same-month-span earlier-date later-date) ) )
```

Finally, here is a version based on approach #2.

```
(define (equal-length-month-span earlier-date later-date)
  (+
    1
    (-
      (days-up-thru later-date)
      (days-up-thru earlier-date) ) ) )

(define (days-up-thru date)
  (+
    (date-in-month date)
    (if (<= (month-number (month-name date)) 8)
      (* (- (month-number (month-name date)) 1) 31)
      (+ 217 (* (- (month-number (month-name date)) 8) 30)
```

Note that you didn't need to make a special case for December. Any legal December date will have 28 or fewer days, so it does no harm to pretend that it has 30 days.

This part was worth 6 points. Some common deductions were the following.

–1 Using `first/butfirst` instead of `month-name` and `date-in-month`, or treating 30 and 31 as two different but essentially equivalent cases instead of calling `days-in-month`. (You were told to use procedures already defined in the version 1 code wherever possible. Many students lost 1 or 2 points here.) However, you were not penalized for using a longer-than-necessary expression involving version 1 procedures, e.g.

```
(+ 1 (- (days-in-month (month-name date)) (date-in-month date)))
```

instead of `(days-remaining date)`.

–1 Off-by-one computation.

–1 Forgetting to apply `month-name` to a date before passing it to `month-number` or `days-in-month`.

–2 In the version based on approach #2, using a big `cond` instead of `month-number` and `days-in-month`.

Missing parts of the computation lost you at least 2, for example, neglecting to count the contribution of the end months, or computing the number of intervening months, but forgetting to multiply the number of intervening months by the month length. (Not dealing with the intervening months at all lost you 4 points.)

Part b was to explain why the revised association of months and days make it easier to finish the first version of the `day-span` code. The first version could not concisely determine the number of days between two arbitrary months. In the reorganized calendar, however, it can, because of the regularity of dates. In particular, the number of days in intervening months can be computed, either as we did it in part a or, in case the end months have different lengths, there will be some months of length 30 (up through July) and some of length 31 (August or later).

Many of you lost 1 point by providing an explanation that assumed that the end months have the same length, as in part a. The full `day-span`, however, needs to handle end months of different lengths.

Simply saying that all the long months preceded all the short months, or that the program now works using the three new procedures, earned you no credit. Some of you lost a point or two for a description that wasn't sufficiently detailed; e.g. you had to say something about the program, not just provide an abstract explanation.

Problem 5 (5 points)

This problem asked you to produce test cases for a given procedure. The versions all involved an "is legal?" procedure with a single argument, checking for a two-word sentence, one of whose words is an integer in a given range. Here are eleven categories for test arguments:

- something that's not a sentence;
- an empty sentence;
- a one-word sentence;
- a sentence containing three or more words;
- a sentence whose symbolic word (the word that's not an integer) is incorrect;
- a sentence in which what should be an integer isn't an integer;
- a sentence in which the integer element is too big;
- a sentence in which the integer element is too small;
- a sentence in which the integer element is as small as legally allowed;
- a sentence in which the integer element is as large as legally allowed;
- a sentence with the words in reverse order (overlaps two of the above cases, but would provide useful debugging information).

You received ½ point for each test case (and explanation) that didn't overlap any of your other tests. Fractional totals were rounded up.

Some special-case deductions: if your calls used two arguments instead of one, or if you included test cases but not explanations, you could earn at most 3 out of 5. If you did both these things, you could earn at most 2 out of 5.

You only received credit for one of a pair of tests for one-word sentences, even if one sentence contained a number and another didn't. Test arguments like `(EE (101))` or `(#t 101)` were treated like any other non-sentence. Commonly overlooked test cases included the following:

- the empty sentence;
- an argument that's not a sentence;
- a number that's not an integer.

Many people focused on the number part of the argument and not on the other areas, such as length, data type, etc.

Problem 6 (7 points)

This problem was the same in all versions. You were to fill in base cases for a `precedes?` procedure that, given two words (`word1` and `word2`) and a sentence (`sent`) as arguments, should return true when `word1` occurs before `word2` in `sent`. We announced at the exam that `precedes?` should return `#f` if the two words are the same, and that you were allowed to assume that the words in the sentence argument are all different.

Here's the most common solution.

```
(define (precedes? word1 word2 sent)
  (cond
    ((empty? sent) #f)
    ((equal? (first sent) word1)
     (member? word2 (bf sent)) )
    ((equal? (first sent) word2) #f)
```

The last clause isn't necessary, because the recursion will continue until the sentence is empty if we omit it. The assumption that the words in the sentence are all different combined with the `member?` call handles the case where the two word arguments are identical. You can replace the call to `empty?` by checking that `sent` contains both `word1` and `word2`; this also simplifies the result when we find `word1`.

```
(define (precedes? word1 word2 sent)
  (cond
    ((equal? word1 word2) #f)
    ((or
       (not (member? word1 sent))
       (not (member? word2 sent)) )
     #f) ; includes the case when sent is empty
    ((equal? (first sent) word1) #t)
    ; word2 is in sent, so it must follow word1
```

This was worth 7 points, 2 for checking for an empty sentence (or for both words in `sent` as just described) and 5 for the rest. Deductions included the following:

–1 All (collective) parenthesis errors.

–1 Checking for empty *words*. An empty word is a perfectly reasonable sentence element.

–1 Simple substitution errors, e.g. `and` for `or` or vice versa, `(empty? (bf sent))` instead of `(empty? sent)`. (The latter was particularly common.)

–1 Returning something like "false" or `( )` instead of `#f`.

–1 Returning `#t` when the word arguments are equal. This often resulted from the `cond` clause `((equal? (first sent) word1) (member? word2 sent))`.

–2 No empty check.

–2 Works only if `word1` and `word2` occur *consecutively* in `sent` (quite common).

–3 After a small correction, works only if `word1` and `word2` occur *consecutively* in `sent`.

–4 Always returns `#t` when `(equal? (first sent) word1)` and `word2` in `sent` hasn't been checked. (–3 if accompanied by a check for identical word arguments.)

–5 Checks neither that `(equal? (first sent) word1)` nor that `(member? word2 sent)`. (–4 if accompanied by a check for identical word arguments.)

Extra base cases were not penalized (provided they didn't produce incorrect behavior). Another common error was to call `first` or `butfirst` without first checking for an empty sentence.