CS 3 (Clancy) **Solutions and grading standards for exam 2**
Spring 2004

Exam information

77 students took the exam. Scores ranged from 6 to 30, with a median of 23 and an average of 22.2. There were 42 scores between 23 and 30, 22 between 16 and 22, and 10 between 6 and 15. (Were you to receive a grade of 23 on both your midterm exams, 46 on the final exam, plus good grades on homework and quizzes, you would receive an A–; similarly, a test grade of 16 may be projected to a B–.)

There were two versions of the exam, A and B. (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 the entire exam (even if the only error is a mistake in adding up your points).

Solutions and grading standards

Problem 0 (2 points)

You lost 1 point on this problem for each of the following:

- you earned some credit on a problem and did not put your name on the page,
- you failed to indicate who you were sitting next to, or
- you did not indicate your lab section or t.a.

The reason for this apparent harshness is that exams can get misplaced or come unstapled, and we would like to make sure that every page is identifiable. We also need to know where you will expect to get your exam returned. Finally, it is sometimes useful to know where particular students were sitting.

Problem 1 (7 points)

In part a, you were to write a procedure named winning-moves that, given a Tic-Tac-Toe board represented as a word of X's, O's, and underscores, along with a player, returns a sentence containing all the winning moves for the player. Your solution was to use procedures already provided in the Tic-Tac-Toe program where possible; you were told at the exam not to worry about duplicate moves. In part b, you were to say which sentence would be returned for the board _O_ _XXOOX (version A) or OOX_XX_O_ (version B).

The simplest solution started by converting the board argument into triples, using the find-triples procedure. Each triple is checked for a potential win using the my-pair? procedure, and only the winning triples are kept. Then the X's or O's are removed from each winning triple, leaving the moves. Here's the code.

```
(define (winning-moves board me)
   (every
      (lambda (triple) (keep number? triple))
      (keep
         (lambda (triple) (my-pair? triple me))
         (find-triples board) ) ) )
```

A slightly more complicated variation was

```
(define (winning-moves board me)
   (accumulate
      sentence
      (keep
         number?
         (accumulate
            word
            (keep
               (lambda (triple) (my-pair? triple me))
               (find-triples board) ) ) ) ) )
```

Part a was worth 5 points, split roughly 3 points for isolating the triples that contain a winning move and 2 for isolating the winning moves from among those triples. The following were common errors that each received a 1-point deduction.

- Omission of the outermost every, as in

  ```
  (define (winning-moves board me)
     (keep
        number?
        (keep
           (lambda (triple) (my-pair? triple me))
           (find-triples board) ) ) )
  ```

  A variation on this error was substitution of the outer every with a keep.

- Adding an extra every, which produced empty words along with move numbers in the return sentence.
- Forgetting the outermost accumulate in the complicated version above.
- Reimplementing my-pair? instead of calling it directly.
- Returning #f instead of an empty sentence.
- Using a content-free name for a procedure or placeholder.
- Substitution of i-can-win? for my-pair?.

Some errors for which more points were deducted were the following.

- Forgetting to call find-triples. This lost 2 points.
- Misuse of i-can-win?, as in

  ```
  (define (winning-moves board me)
     (keep
        (lambda (triple) (i-can-win? triple me))
        (find-triples board) ) )
  ```

  The flaw here is that i-can-win? takes a sentence of triples as argument, not a single triple. This solution would have received a 2, losing 1 point for isolating the winning triples and 2 points for isolating moves within those triples.

- Using recursion. You lost all the points for the part (isolating winning triples, or isolating individual moves) you used it in.

The answer to part b is (4 3 1) on version A and (4 9 7) on version B. You could earn points on this part only if your solution to part a was complete enough to distinguish among the possibilities. The answer and the explanation were each worth 1 point, for a total of 2 for this part.

One correct explanation merely traces the calls to find-triples and the higher-order procedures. In version A, for example:

1. Find-triples returns (1o3 4xx oox 14o oxo 3xx 1xx 3xo) for the given board.
2. The outer keep returns (4xx 3xx 1xx).
3. The every returns (4 3 1).

Problem 2 (8 points)

This problem involved providing a single Roman numeral that would distinguish all possibilities for omitting a call to butfirst in the roman-sum procedure, then determining the result of supplying that value to decimal-value using three different buggy versions of roman-sum. The two exam versions differed in the last version of roman-sum.

Here's what the various tests in roman-sum check for.

test	*circumstances*
`(empty? number-sent)`	succeeds for a Roman numeral ending in a prefix/prefixed pair
`(empty? (bf number-sent))`	succeeds for a Roman numeral ending in an unprefixed digit
`(not (starts-with-prefix? number-sent))`	succeeds for an unprefixed digit not at the end of the Roman numeral
`(starts-with-prefix? number-sent)`	succeeds for a Roman numeral containing a prefix/prefixed pair

The last three cond clauses are the ones that may contain the bug. Complete testing with one test case would then involve a Roman numeral that

- contains a prefix/prefixed pair;
- ends with an unprefixed Roman digit;
- contains an unprefixed Roman digit earlier in the Roman numeral.

Examples are xlii and cxli. Many of you provided a Roman numeral that ended in a prefix/prefixed pair, which wouldn't distinguish a correct version of roman-sum from a version missing the butfirst call in line 4.

Here are the various buggy versions of roman-sum, together with their behavior.

version of roman-sum	behavior

`(define (roman-sum number-sent)`
`  (cond`
`  ((empty? number-sent) 0)`
`  **((empty? number-sent) (first number-sent))**`
`  ((not (starts-with-prefix? number-sent))`
`...)`
`  ((starts-with-prefix? number-sent) ...) )`
`)`
	Normal behavior for a Roman numeral ending in a prefix/prefixed pair; crash in second call to first in starts-with-prefix? when checking the last digit in a Roman numeral ending in an unprefixed digit.

`(define (roman-sum number-sent)`
`  (cond`
`  ((empty? number-sent) 0)`
`  ((empty? (bf number-sent)) ... )`
`  ((not (starts-with-prefix? number-sent))`
`  **(+ (first number-sent)**`
`  (roman-sum number-sent)) )`
`  ((starts-with-prefix? number-sent) ... ) )`
`)`
	Infinite recursion results from an unprefixed Roman digit that's not at the end of the Roman numeral.

`(define (roman-sum number-sent)`
`  (cond`
`  ((empty? number-sent) 0)`
`  ((empty? (bf number-sent)) ... )`
`  ((not (starts-with-prefix? number-sent))`
`... )`
`  ((starts-with-prefix? number-sent)`
`  (+`
`  **(- (first number-sent) (first number-sent))**`
`  (roman-sum (bf (bf number-sent))) ) ) )`
`)`
	(version A only) The value of any prefix/prefixed digit pair is ignored.

`(define (roman-sum number-sent)`
`  (cond`
`  ((empty? number-sent) 0)`
`  ((empty? (bf number-sent)) ... )`
`  ((not (starts-with-prefix? number-sent))`
`... )`
`  ((starts-with-prefix? number-sent)`
`  (+ ...`
` **(roman-sum (bf number-sent))))))**
	(version B only) Any prefixed digit is counted twice in the sum.

A single score (maximum 8 points) was given on this problem, rather than individual scores for the two parts. Scoring proceeded as follows.

1. First, we deducted points for an inadequate test case:

   –1 for a Roman numeral with both prefix/prefixed pairs and unprefixed digits that ends in a prefix/prefixed pair, since this would not distinguish the first buggy version from a correct version;

–3 for a Roman numeral with only prefix/prefixed pairs (e.g. xliv or iv), which would produce the correct value for versions of roman-sum missing the first or second butfirst call;

–3 for a longer Roman numeral with no prefix/prefixed pairs, which would fail to reveal omission of any of the butfirst calls in the starts-with-prefix? cond clause;

–5 for a single Roman digit.

2. If you received a deduction of more than 1 point here, it's because two of the buggy roman-sum procedures produce the same answers. We deducted an additional point if you had different answers for roman-sum procedures that should be producing the same answer.
3. We then deducted 1 point for each wrong result, after choosing the best explanation among those that should be duplicates. Where your test case produced a crash, you could lose 2 points; you needed both to note the crash and to say that it happened in starts-with-prefix?.

Here is a grading example. Suppose your test case was iv. In step 1 you lost 3 points. Your results should have been 4, 4, and 0 (version A) or 9 (version B). If your first result didn't match your second, you lost 1 more point in step 2. If neither your first result nor your second was 4, you lost 1 point in step 3; you also lost 1 point if your third result was incorrect.

As noted above, the most common error was a test case that ended in a prefix/prefixed pair. Another common error was a vague or incomplete specification of where a crash would occur; this lost 1 point. (Some of you seemed to think that the crash would occur in the ((empty? number-sent) (first number-sent)) clause, even though an empty number-sent would have caused roman-sum to return in the clause immediately preceding.)

Problem 3 (5 points)

In this problem, you analyzed a call to accumulate that used a buggy max procedure:

```
(define (max x y)                ; incorrect version
   (if (> x x) x y) )
```

As you noted in lab, accumulate used with max and, say, the sentence (3 1 4 2) works as follows.

```
(max 3 (max 1 (max 4 2)))
```

The max procedure always returns its second argument, since x is never greater than itself. Thus accumulate used with this max procedure would always return the last value in the sentence argument. Its result would be the correct one exactly when the last value is the largest in the sentence argument.

That explanation is essentially what we wanted for part c. This part was worth 3 points, 2 for the answer and 1 for the justification. We didn't require that you specify the order of accumulation, since the opposite order would produce the same behavior.

Parts a and b were worth 1 point each, with no partial credit. The two versions differed in part a: in version A, you evaluated (sent-max '(1 3 2))—the result is 2—while in version B, you evaluated (sent-max '(2 3 1))—result is 1. You lost the part a point if you gave a sentence instead of a number.

If you erroneously noted that the buggy max always returned its first argument instead of its second, you earned no points for this problem.

Problem 4 (8 points)

For this problem, you were to write a procedure split that split its word argument into subwords according to the lengths specified in its second argument. This problem was the same on both versions. Here are some solutions.

Word-by-word solution

```
(define (split longWord lengths)
   (if (empty? lengths)
      (sentence longWord)
      (sentence
         (first-n longWord (first lengths))
         (split (butfirst-n longWord (first lengths)) (bf lengths)) ) ))

(define (first-n wd n)
   (if (= n 0) "" (word (first wd) (first-n (butfirst wd) (- n 1)))) )

(define (butfirst-n wd n)
   (if (= n 0) wd (butfirst-n (butfirst wd) (- n 1))) )
```

Character-by-character accumulating solution

```
(define (split longWord lengths)
   (helper longWord lengths "" '( )) )

(define (helper longWord lengths cur-word so-far)
   (cond
      ((empty? lengths) (sentence so-far longWord))
      ((= (first lengths) 0)
       (helper longWord (bf lengths) "" (sentence so-far cur-word)) )
      (else
       (helper
         (butfirst longWord)
         (sentence (- (first lengths) 1) (butfirst lengths))
         (word cur-word (first longWord))
         so-far) ) ) )
```

Two more character-by-character solutions

```
(define (split longWord lengths)
   (cond
      ((empty? lengths) (sentence longWord))
      ((= (first lengths) 0) (sentence "" (split longWord (bf lengths))))
```

```
      (else
         (sentence
            (word                                          ; make a new first word
               (first longWord)
               (first
                  (split
                     (butfirst longWord)
                     (sentence (- (first lengths) 1) (butfirst lengths)) ) ) )
            (butfirst
               (split
                  (butfirst longWord)
                  (sentence (- (first lengths) 1) (butfirst lengths)) ) ) ))) )
(define (split longWord lengths)
   (cond
      ((empty? lengths)
       (sentence longWord))
      ((= (first lengths) 0)
       (sentence "" (split longWord (butfirst lengths))))
      (else
       (helper
         longWord
         (split
            (butfirst longWord)
            (sentence (- (first lengths) 1) (butfirst lengths)) ) ) ) ) )
(define (helper longWord recursiveResult)
   (sentence
      (word (first longWord) (first recursiveResult))
      (butfirst recursiveResult) ) )
```

In the word-by-word and the accumulating character-by-character solutions, up to 2 points was awarded for the base case and 3 for each recursive case. In the other two character-by-character solutions, the base case and the "(= (first lengths) 0)" case were both worth 2 points and the else case was worth 4. Common errors with corresponding deductions were as follows.

- 2 points (all the base case points) were deducted for a base case of

  ```
  ((empty? lengths) "")
  ```

  or

  ```
  ((empty? lengths) '( ))
  ```

  that is, something not involving longWord.

- 1 point was deducted for a base case of

  ```
  ((empty? lengths) longWord)
  ```

  that is, where a word was returned instead of a sentence.

- 1 point was deducted for using an empty sentence where an empty word was required, or vice versa.

There were numerous instances of word/sentence confusion. For example, some solutions would make a recursive call with just the first of a sentence of numbers. Another symptom of this confusion was a return sentence containing all the individual characters from longWord as its words.

Other students attempted an accumulating recursion but *replaced* the accumulator rather than adding to it at every step.

Solutions taking the word-by-word approach and thereby taking advantage of structure provided in the problem were much more likely to be correct. (Examples of this approach that you saw in lab were the no-vowels, sorted, occurs-in?, and thoroughly-reversed procedures in the third week of recursion.) In particular, no one provided a correct version of the last two character-by-character solutions above.

You were not allowed to use higher-order procedures, in particular repeated. Use of a higher-order procedure lost you all the points for the part you used it in.