CS3L, Summer 2011 Exam 2 Key

Question 1:

This one ended up being a real killer. Perhaps it's because it was the very first question on the exam, and so some people might have assumed it was easier than it really was and not thought it through entirely; perhaps it's because the problems with the code are fairly subtle, and tracing code by sight, on paper, is inherently tough. Only a couple people got the full 8 points; most people earned between 2 and 6.

The mysterious "The above procedure never crashes, and always produces the correct result for any inputs" sentence about `substitute` was supposed to clarify that you weren't supposed to think about weird crashes from malformed inputs or other bizarre scenarios, and to get you thinking about different possible inputs.

There's nothing unsound about moving the `if` in `substitute2` to the inside of the call to `se`, but Nitpickerson (a recurring character in classes I've taught) has carelessly switched the order of the two arguments to `se`. This causes `substitute2` to produce what *would* be the correct answer... if it weren't in reverse order, that is! (Our example would come out as `(man no is man that)`.) But sometimes the reversed expression is still correct – what if the original sentence is empty, or has only one element, or reads the same forwards and backwards? So the answer is **B**, even though the procedure will almost always produce the wrong answer, and will only be right "by accident" in a handful of cases.

The problem with `substitute3` is that the call to `(substitute3 (bf sent) from to))` in the `let` will be evaluated before the body is evaluated (as must always be the case in order for a `let` to work). Then that call will do the same thing, and so on; the body of a `let` will never be reached. You might think this would cause the procedure to run forever, but it still makes "progress" each time because of the `bf`, and so it will eventually crash with that familiar "invalid argument" error that you get when you try to take the `first` or `bf` or `car` or whatever of something empty or too short. So the answer is **G**.

Scoring: 4 points for each question, with partial credit awarded for answer choices that were clearly on the right track or otherwise had some merit (subjective, of course, but the decisions are final). Each question had a "second best" answer: C for `substitute2`, and E for `substitute3`; these were worth 2 points. Some other answers for each question were worth 1 point.

Question 2:

`fib` trace:

```
(fib 5) =>
(+ (fib 4) (fib 3)) =>
(+ (+ (fib 3) (fib 2)) (+ (fib 2) (fib 1))) =>
(+ (+ (+ (fib 2) (fib 1)) 1) (+ 1 0)) =>
(+ (+ (+ 1 0) 1) (+ 1 0)) =>
(+ (+ 1 1) 1) =>
(+ 2 1) => 3
```

There are a total of **nine** calls (the initial call and eight recursive calls).

`fib2`: As the problem statement explains, this procedure starts off with the first Fibonacci number (0) as `smaller` and the second Fibonacci number (1) as `larger`, and generates each subsequent Fibonacci number by adding the previous two (`smaller` and `larger`) together. This new number becomes the new `larger`, and the old `larger` becomes the new `smaller`. Add 1 to `currentnum` to mark your progress (and correctly reflect which number we're on).

Once the `n`th Fibonacci number has been generated (and we'll know this because `currentnum` will equal `n`), you need to return it – but what is it? It's the `smaller` of the two. One way to see this: what should happen when you type `(fib2 1)`? You should get back 0, since 0 is the first Fibonacci number... and 0 starts off in `smaller`. (A lot of people missed this, perhaps because the first and second Fibonacci numbers are often defined as 1 and 1 instead of 0 and 1... but you had to be consistent with the description given in the problem, which is the mathematically preferred / "correct" definition.)

Some of you used `(- n 1)` or `(- n 2)` in place of `n`; even if you made the rest of your answer consistent with this, it had the unfortunate side effect of making `(fib2 1)` (or even `(fib2 2)`) run forever. (That was only a one point deduction, though)

```
(define (fib2 n)
  (define (fib2helper smaller larger currentnum)
    (if (= currentnum n)
        smaller
  (fib2helper larger
              (+ smaller larger)
              (+ currentnum 1))))
  (fib2helper 0 1 1))
```

Process types:

`fib` is a **recursive** / expanding process, as the expanding expression in the trace above demonstrates. `fib2` is an **iterative** / non-expanding process. It always results in a call to itself that isn't part of a growing expression. The "progress" is made by giving fib2helper different arguments, not by adding to the eventual result of evaluating a big nested chain of expressions (as would be the case for a recursive process).

Scoring: 2 points for the number of calls (1 if you wrote 8, presumably by forgetting the original call); 1 point per blank for fib2helper; 1 point per choice for the process types.

Question 3:

`is-subset?` is similar to the other "step through these two sentences, making exactly one pass through each" problems we've looked at (`intersection`, `union`, `merge`). In this case, we only move on in `s1` when we've verified that the first of `s1` is in `s2`.

Think about the `cond` clauses in order:

* If `s1` is ever empty, then either it was the empty sentence to begin with (and is trivially a subset of `s2`, as per the examples), or we've found all the elements of `s1` in `s2`. So this case should return `#t`.

* If `s1` is not empty (which must be the case if we got past the `(empty? s1)` of the `cond`) and `s2` is empty, then we have at least one thing in `s1` that we're never going to find in `s2`. So this case should return `#f`.

* If the `first` of `s1` is smaller than the first of `s2`, then we're never going to find the `first` of `s1` in `s2`, because, since the sentences are both sorted, everything else in `s2` is even larger! So this case must return `#f`. If you were relying too heavily on past problems like this, you might have *reeeally* wanted to put a recursive call here (this was a moderately common error), but it's actually a base case.

* If the `first` of `s1` is larger than the `first` of `s2`, then throw out the `first` of `s2` (it can't possibly match anything in `s1`, because everything in `s1` is larger than it) and call `is-subset?` again.

* Otherwise, the `first` of `s1` is equal to the `first` of `s2`. This bodes well for `s1` being a subset of `s2`! Advance past both of these items and keep checking.

```
(define (is-subset? s1 s2)
  (cond ((empty? s1) #t) ; clause I
        ((empty? s2) #f) ; clause II
        ((< (first s1) (first s2)) ; clause III
         #f)
        ((> (first s1) (first s2)) ; clause IV
         (is-subset? s1 (bf s2)))
        (else
         (is-subset? (bf s1) (bf s2)))))
```

`cond` clauses III and IV are switched with no ill effects, because the test conditions are mutually exclusive: `(first s1)` can't be both greater than *and* less than `(first s2)`.

If `cond` clauses II and III are switched, we try to take the first of `s2` without first checking that `s2` isn't empty, which will often cause a crash.

If `cond` clauses I and II are switched, a problem arises: the case `(is-subset? '() '())` no longer produces the correct answer. (This was subtle, but `(is-subset? '() '())` *was* one of the example cases...)

Scoring: 7 points for the blanks (e.g., -2 for a bad recursive call in blank 3, generally -3 instead of -4 for two errors), 1 point per choice for the `cond` clause swap part.

Question 4:

The joke here was supposed to be about inebriated / otherwise delirious texting – I liked the idea of pouring out one's heart to STk in a moment of weakness and then not remembering it later – but I don't think that came through all that well.

Here's how to deal with `(list 1 (append (cons '(2) '()) (cons '() (list 3))))`:

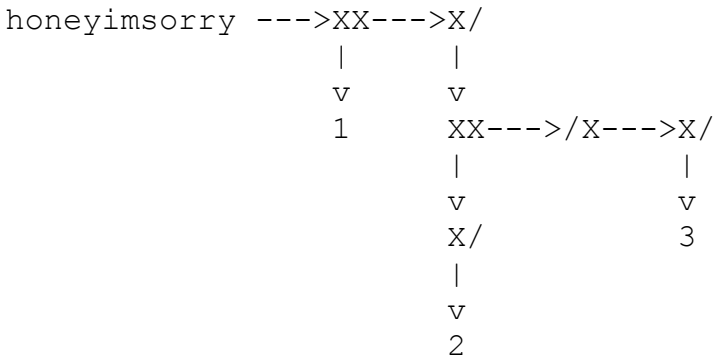
`'(2)` is a list containing one item, `2`: `(2)`
`(cons '(2) '())` produces a pair, the first half of which points to the aforementioned `(2)`, and the second half of which is the empty list. So this is really a one-item list in which the one item is itself a list. We get: `((2))`

`(list 3)` produces a list containing one item, `3`: `(3)`
`(cons '() (list 3))` sticks `()` onto the start of that list, as a new item: `(() 3)`. The empty list doesn't always just "go away" automatically! Here, it's a full-fledged item in a list, not the end of a list.

`(append (cons '(2) '()) (cons '() (list 3)))` takes the end of that one-item list, `((2))`, and (more or less) makes it point to the other list, `(() 3)`. The result is `((2) () 3)`.

`(list 1 <otherstuff>)` makes a new two-item list in which the first item is `1` and the second is the previously calculated `((2) () 3)`. So the overall answer is **`(1 ((2) () 3))`**.

You didn't have to use a box-and-pointer diagram to get this, but I maintain that they really are helpful for checking your work even if you can just eyeball it all out without the diagram. Here's the diagram. An X stands for a half of a pair, and a / stands for an empty list. (no strikes and spares here!)



Many of you did draw adorable farm animals (or otherwise endearing artwork), and I meant to go back and comment on them, but it slipped my mind. The drawings were much appreciated, though.

To get `1`, `2`, and `3` out of this list, just use the above box-and-pointer diagram; a `car` is a move downward and a `cdr` is a move to the right. And make sure you don't get the letters backwards; the *rightmost* letter is applied first.

1: **`(car honeyimsorry)`**
2: **`(caaadr honeyimsorry)`**
3: **`(caddadr honeyimsorry)`**. This wouldn't work in STk, because `cxrs` with five or more `a`s and `d`s aren't defined, but it was fine for this question. In STk, you'd type `(caddar (cdr honeyimsorry))`, or one of the other equivalent expressions.

Scoring: 4 points for the list as Scheme would print it, minus 1 point for every set of parentheses I had

to add or remove. 1 point per `cxr` question, for either the right answer or an answer consistent with a wrong answer to the list part.

Question 5: (dropped from the exam)

There was nothing *wrong* with this question, but it involves an aha moment (either you see how to do it in an elegant way, or you're stumped and maybe spend a lot of time on a less straightforward solution), so we cut it... and we're glad we did, since there was some time pressure even without it! (Can you figure out what the original theme of this problem was? It has been sanitized a bit.)

This problem might have reminded you of the moving average problem from Lab 13, in that it asks you to move a 3-unit window along a sentence and do something to the central element of those 3 units. You could have gleaned a nice trick from that problem: *pad* the voter data with a fake empty booth on each end. This doesn't create any new calm voters that weren't there before (since adding empty booths doesn't remove voters who are making voters on the edge nervous), and it frees you from the burden of treating the edges of the voting booth row differently.

This problem is, to borrow a movie title, all about EVE. A voter will be calm if and only if he has an E on either side, so all you have to do is step through the sentence looking for EVEs and counting them up. You have to be careful to stop before you run out of sentence, though; once there are fewer than three elements, stop (since your "window" is centered on an empty fake "padding" booth at that point anyway).

```
(define (num-calm layout)
  (define (helper remaining)
    (cond ((< (count remaining) 3) 0)
          ((and (equal? (item 1 remaining) 'E)
                (equal? (item 2 remaining) 'V)
                (equal? (item 3 remaining) 'E))
           (+ 1 (helper (bf (bf remaining)))))
           ; can safely move 2 because we know the next isn't V
          (else (helper (bf remaining)))))
  (helper (word 'E layout 'E)))
```

Question 6:

```
(define (ways-home t h)
  (define (helper currpos timeleft)
    (cond ((= currpos h) 1) ; if she's home, we've found a
                            ; solution
          ((= currpos 0) 0) ; back at the club - will never get home
          ((= timeleft 0) 0) ; this check has to be after the
                             ; "are we home" check
          ; otherwise, add the results of the solutions starting
          ; with each of the possible 2 moves from here
          (else (+ (helper (+ currpos 1) (- timeleft 1))
                   (helper (- currpos 1) (- timeleft 1))))))
  (helper 1 t))
```

This question originally included a policeman (standing somewhere between the starting position and home) who would get increasingly suspicious every time Ke$ha walked by, and would eventually arrest her, preventing her from getting home. But the policeman was removed as part of a general drive to uncomplicate the exam.

This question really should have been called “Tick Tock, Random Walk”. A missed opportunity! (Random walks really are sometimes called “drunkard's walks”, too.)

Scoring: Graded by Eric.

Some tests:

```
(ways-home 10 8) => 7
(ways-home 11 8) => 33
(ways-home 11 5) => 33
(ways-home 11 6) => 58
```

Rubric:
8 pts: Perfect
7 pts: Perfect, but with trivial mistakes
6 pts: Mostly right:
- You checked the time base-case before the are-you-home? base case (would mean that `(ways-home 2 3)` would return 0, when it should return 1)
- You slightly messed up one of the base cases (say an off-by-one error)

4 pts: Almost right
Pretty much had the recursion 'form' down, (including checking for time, being-at-home) but:
- You didn't get the 'club' base-case correct. Usually happened if you didn't keep track of your current position.
- You screwed up a check (i.e. you did a bunch of checks concerning time, but didn't do something for when (= t 0)

2 pts: A start
Better than a 1 - has much of the logic, but screws up say the recursion or something.
1 pts: A start
Presence of correct branching recursion, but nothing much else. has some recursive procedure, with some attempt to do a thing

Question 7:

The first part was supposed to make you think of and use `assoc`, which is really the most efficient way of getting a student's data out of the gradebook. Then pull out the list of grades from the student's record; that record is a list, and the list of grades is the second element, so use `cadr` to get it. (`cdr` would return everything but the first element, *not* the second element. Finally, use `accumulate` (the list version, with three arguments including a base case) to add up all the results. I also accepted the sentence version of `accumulate`, with only two arguments (no base case), since it *does* work... or even `reduce` (but only with two arguments, since it won't take three).

```
(define (glookup login gb)
  (define (glookup-helper who)
    (accumulate + 0 (cadr (assoc who gb))))
(* (/ (glookup-helper login) (glookup-helper 'pp)) 100))
```

Scoring: 1 point per blank; leaving off the base case of 0 (and thus using the sentence version of accumulate) was OK. 1/5 for solutions that disregarded the instructions and filled in much more than one atom per blank, even if the solution might have worked. This was supposed to be an explicit test of your knowledge of list HOFs.

`add-assignment`: We have no way of mutating lists in CS3L, so you had to build up a new gradebook, using the old one as a template. The requirement that a gradebook be in alphabetical order was supposed to force you to go through the old gradebook in its existing order and (re)build it that way; you couldn't go through `scores` in that order (which might not have been alphabetical), and you wouldn't have wanted to anyway, since it would have made it a pain to find the students who weren't in `scores`. After that, it was mostly a question of getting the list selectors / constructors right, and correctly handling missing students.

```
(define (add-assignment scores gb)
  (map (lambda (student)
         (let ((score (assoc (car student) scores)))
           (if (not score)
               (list (car student)
                     (append (cadr student) (list 0)))
               (list (car student)
                     (append (cadr student) (cdr score))))))
       gb))
```

Recursively:

```
(define (add-assignment scores gb)
  (cond ((null? gb) '())
        ((assoc (caar gb) scores)
         (cons (list (caar gb)
                     (append (cadar gb)
                             (cdr (assoc (caar gb) scores))))
               (add-assignment scores (cdr gb))))
        (else
          (cons (list (caar gb) (append (cadar gb) (list 0)))
                (add-assignment scores (cdr gb))))))
```

Scoring: Graded by Eric.

8 pts - Perfect
7 pts - Perfect, but with trivial mistakes (say, entire result is in an inner list, everything is right but null-check is out of order, etc)
6 pts - Mostly right, the output just has weird nestings/dots
5 pts - Mostly right: We look at every student in `gb`, the output is

'corrupted', i.e;

```
> (add-assignment '((db 8) (pp 10)) my-grades)
        ((da (4 2 da 8)) (db (2 2 0)) (pp (5 3 pp 10)))
```

or some other flavor (say, each record is out-of-order, like:

```
((8 (da 4 2)) ...)
```

4 pts - The student looks at each student in `gb`, and tries to create an updated entry for student (where we decide to input a score if present, or 0 if not present in scores). However, an error occurs during execution (say, took the `car` of an empty list, or we take the `cdr` of `#f`), or we lose the student's name/grades in the process.

2 pts - We at least look at each student in `gb`, and try to do something to each student.

0 pts - Nothing

Question 8:

`from-broman:`

```
(define (from-broman numeral o r bro)
  (cond ((empty? numeral) 0)
        ((and (>= (count numeral) 3)
              (equal? (item 1 numeral) 'B)
              (equal? (item 2 numeral) 'R)
              (equal? (item 3 numeral) 'O))
         (+ bro (from-broman (bf (bf (bf numeral))) o r bro)))
        (else
         (+ (cond ((equal? (first numeral) 'B) 1)
                  ((equal? (first numeral) 'R) r)
                  (else o))
            (from-broman (bf numeral) o r bro)))))
```

The strategy for this problem was nearly the same as the strategy for `from-roman` from HW4. Just walk through the numeral, converting every letter to its value, but detect and convert BROs appropriately (and then step all the way past them). The tricky part is checking for BRO (a check which has to come before, or otherwise take into account, the check for just a plain B). Quite a few people forgot to check that the remaining numeral was at least three letters long before checking, e.g., the third `item` or, equivalently, the `(first (bf (bf )))`. ddd

Scoring: I typed in everyone's code, debugged it, and checked it against several cases in STk. Most solutions had enough of the right idea to warrant only minor deductions.

8 points: Perfect
7 points: Minor but non-negligible error (missing parentheses didn't count unless they were missing / extra left parentheses or made the code behave in weird ways until the parenthesization was fixed)
6 points: Significant error
5 points: Two significant errors
2 points: Has an idea, but too far off

`to-broman:`

This was a real beast (both to code and to grade!) Unlike `from-broman`, **it is not the straightforward equivalent of its HW4 counterpart.**

```
(define (to-broman num o r bro)
  (define (helper curr sofar)
    (cond ((< curr 0) "")
          ((= curr 0) sofar)
          (else (se (helper (- curr 1) (word sofar 'B))
                    (helper (- curr o) (word sofar 'O))
                    (helper (- curr r) (word sofar 'R))
                    (helper (- curr bro) (word sofar 'BRO))))))
  (accumulate
    (lambda (x y)
      (if (or (< (count x) (count y))
              (and (= (count x) (count y)) (before? x y)))
          x
          y))
    (keep (lambda (x) (not (empty? x))) (helper num ""))))
```

The major challenges posed by this problem are:

1. Realize that the "greedy" strategy of taking a number and successively subtracting off the largest possible value won't work in all cases. **The example outputs should have made this clear.** If the greedy strategy really worked, then the very first example, `(to-broman 7 5 10 6)`, would have had a BRO in it, because 6 would have been subtracted off first; the example below that would have been BRO instead of BOR. Also, there was no way to know the relative sizes of O, R, and BRO unless you wrote a helper to examine this or a bunch of conditionals; by not specifying the size order, I hoped to steer you away from this wrong solution path. (But people who tried the greedy strategy generally didn't seem to consider the size order, implicitly assuming that O < R < BRO, or tried to consider it but got bogged down in the details. Only a handful of answers dealt with this correctly.)

2. Realize that therefore, all possible valid Broman representations of the number must be generated, and that this problem is very similar to the Nom / bridge-building / change-making etc. problems.

3. Generate a sentence containing all of these representations. (Some of you used list procedures, and this dramatically increased the probability of going astray. I specifically wrote this problem as a word / sentence problem to spare you the added difficulties of dealing with list selectors and constructors.) Deal with the issue of bad solutions (empty sentences in my version).

4. Go through the solutions and narrow them down to the shortest, and then to the alphabetically earliest. (This can be done simultaneously). Unfortunately, some people wrote this part in a way that would let an alphabetically earlier but longer solution beat out an alphabetically later but shorter one.

Scoring: I typed in everyone's code, debugged it, and checked it against several cases in STk. Many people received scores in the 3-5 range, but there were a good number of 6-10s. There was a definite "tipping point" – your code had to be quite close to correct to receive more than 5 points, and code with multiple errors usually failed to give me the "this is almost right" impression, even if the underlying

idea was sound. This was the “very hard” problem on the test, intended to differentiate between the top ability levels, and so my expectations were high and my grading standards were stricter. Some people obviously ran out of time on this problem, but these exams are also a test of your ability to code rapidly, based on comfort with Scheme syntax and hours and hours of practice, and it would have been unfair to the people who did take the time to get the code exactly or mostly right if I'd given substantial credit for just having the right idea or sketching out a skeleton. Similarly, the incorrect greedy strategy was capped at 5 points because it just wasn't the right answer, no matter how well-implemented. If you believe your solution was graded too harshly, please let me know, but I tried my best to be as consistent as possible, so I'll only change a score if you can point out that I dramatically undervalued your solution (i.e. it was much closer to completely correct than I thought it was).

10: perfect; only a couple people pulled this off
9: perfect, but with a small but non-negligible mistake (e.g., a wrong initial value like `'()` in place of `""`)
8: workable after a significant error was (or a couple minor errors were) fixed
7: workable after a couple errors fixed; or, nearly equivalent to the right solution path, but flawed in some subtle way
6: major progress toward the right solution, but with some big flaw, e.g., doesn't handle selecting the preferred representation out of the sentence of all the possible representations
5: a perfect or nearly perfect implementation of the unsuccessful “greedy” strategy, or an implementation that was on the right track, but with several errors
4: a flawed implementation of the “greedy” strategy, or an implementation that was on the right track, but too full of errors / incomplete / hard to follow
3: a “greedy” implementation with a lot of errors, or a solution that was just too complex / hard to follow and would have needed major surgery to work
1-2: a start, but not much more