CS3L Summer 2011 Exam 2
Time: up to 170 minutes (you may leave when finished; or, you must stop promptly at noon)

Name: ___________________________ Login: cs3-_______

First names of the people to your left and right, if any: Left: ________ Right: ________

1. You may solve coding questions using any combination of recursion and built-in HOFs. You may use helpers.

2. If you have difficulty understanding the language of a question, or exactly what the question is asking, please ask us for clarification as soon as possible.

3. Assume that your procedures will only be given inputs of the form described in the problem statement. You don’t need to anticipate / check for inputs with the wrong number / type of arguments, etc.

4. You must explicitly define any procedures that you use that aren’t built into STk, even if you’ve previously defined them in lab.

<table>
<thead>
<tr>
<th>#</th>
<th>Question name</th>
<th>Possible</th>
<th>Your score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Substitute Substitutes</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Two Ways To Fib</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>IM IN UR SET</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>I Think I STked Last Night</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Poll Positions</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Tick Tock</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Making the Grades</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Do As The Bromans Did</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>80</td>
<td></td>
</tr>
</tbody>
</table>
**Question 1: Substitute Substitutes**

In lab, you wrote the `substitute` function, which replaces any and all instances of a word from in a sentence sent with a new word to:

```
(define (substitute sent from to)
  (cond ((empty? sent) '())
        ((equal? (first sent) from)
         (se to (substitute (bf sent) from to)))
        (else
         (se (first sent) (substitute (bf sent) from to)))))
```

```
> (substitute '(that moon is no moon) 'moon 'man)
(that man is no man)
```

The above procedure never crashes, and always produces the correct result for any inputs.

Quibble Nitpickerson hates typing out `(substitute (bf sent) from to)` in two different places in the program, and he has proposed two different ways to rewrite `substitute` to avoid this. Here’s one of his attempts:

```
(define (substitute2 sent from to)
  (if (empty? sent)
      '()
      (se (substitute2 (bf sent) from to)
          (if (equal? (first sent) from)
              to
              (first sent))))
```

Which of the following best characterizes `substitute2`? **Circle only one.**

A. It will never crash, and it will always produce the correct result.
B. It will never crash, but its result will be correct in some cases and incorrect in other cases.
C. It will never crash, but its result will always be incorrect.
D. It may crash, produce incorrect results, or produce correct results, depending on the inputs.
E. It will always run forever (or at least until it segfaults / coredumps).
F. It will always cause a bad function error.
G. It will always cause an invalid argument error.
H. It will always cause an unbound variable error.
**Question 1, continued**

Here’s another of Nitpickerson’s attempts, in which he uses a `let` to avoid the redundancy:

```
(define (substitute3 sent from to)
  (let ((nextstep (substitute3 (bf sent) from to)))
    (cond ((empty? sent) '())
          ((equal? (first sent) from) (se to nextstep))
          (else (se (first sent) nextstep)))))
```

Which of the following best characterizes `substitute3`? **Circle only one.**

A. It will never crash, and it will always produce the correct result.
B. It will never crash, but its result will be correct in some cases and incorrect in other cases.
C. It will never crash, but its result will always be incorrect.
D. It may crash, produce incorrect results, or produce correct results, depending on the inputs.
E. It will always run forever (or at least until it segfaults / coredumps).
F. It will always cause a bad function error.
G. It will always cause an invalid argument error.
H. It will always cause an unbound variable error.
Question 2: Two Ways To Fib

Let F(n) denote the nth Fibonacci number. F(1) is 0, F(2) is 1, and for n > 2, F(n) = F(n-1) + F(n-2). So F(3) = 1, F(4) = 2, F(5) = 3, F(6) = 5, and so on. In lab, you’ve seen the following procedure for generating the nth Fibonacci number:

(define (fib n)
  (if (<= n 2)
      (- n 1) ; i.e., 0 for n = 1, 1 for n = 2
      (+ (fib (- n 1)) (fib (- n 2)))))

1. Suppose that you type in (fib 5). Including this original call, how many total calls to fib will be made?

   __________________________

2. Fill in the blanks to complete the following alternate procedure for generating the nth Fibonacci number. It starts off “knowing” F(1) and F(2), generates each new Fibonacci number by adding together the two it currently “knows”, and remembers which Fibonacci number it’s currently on. **Do not otherwise add to or alter the existing code.**

(define (fib2 n)
  (define (fib2helper smaller larger currentnum)
    
    (if (= currentnum __________________________)

    __________________________

    (fib2helper larger

    __________________________

    __________________________))

  (fib2helper 0 1 1))

3. Both fib and fib2helper are recursive procedures, but which one(s), if any, generate a recursive process? Circle any and all that apply:

   fib   fib2helper
**Question 3: IM IN UR SET**

The following procedure, `(is-subset? s1 s2)`, takes two sentences `s1` and `s2` and returns `#t` if every item in `s1` is also in `s2`, and `#f` otherwise. Each sentence contains only numbers, in sorted order (from smallest to largest), and no number is duplicated within a sentence. Here are some examples of the intended behavior:

```
STk> (is-subset? '(2 4) '(1 2 3 4 5))  #t
STk> (is-subset? '(2 5) '(2 3 4))     #f
STk> (is-subset? '() '(1 2 3 4))      #t
STk> (is-subset? '(1) '())            #f
STk> (is-subset? '() '())             #t
STk> (is-subset? '(1 2 3) '(1 2 3))   #t
```

1. Fill in the blanks to complete `is-subset?`. **Do not otherwise add to / alter the existing code.**

```
(define (is-subset? s1 s2)
    (cond ((empty? s1) ______________________) ; clause I
          ((empty? s2) ______________________) ; clause II
          ((< (first s1) (first s2)) ; clause III
            ________________________________)
          (>= (first s1) (first s2)) ; clause IV
            ________________________________)
          (else
            ________________________________))))
```

2. The `cond` clauses are currently in the order I, II, III, IV. Which of the following orderings for the first four `cond` clauses in `is-subset?` would create the potential for an error (crash) or a wrong answer? (A single “`cond` clause” includes both the test and the outcome of the test, not just the test.) **Circle all that apply.**

```
I, II, IV, III    I, III, II, IV    II, I, III, IV
```
Question 4: I Think I STked Last Night

You wake up after a party to find that you apparently typed the following into STk. Must have been a wild night!

STk> (define honeyimsorry
       (list 1 (append (cons '(2) '()) (cons '() (list 3)))))
honeyimsorry

1. What does Scheme print when you type honeyimsorry at the prompt? (The expression does not cause an error, and the output does not have any dots. in it -- only numbers and parentheses.)

The following space is for any box and pointer diagrams that you choose to draw. You will not be graded on what you draw here. You can draw an adorable farm animal if you so desire.

2. Fill in combinations of car and cdr (represented as shorthand like caddar) that would make STk print/return 1, 2, and 3, respectively. (Be careful; returning (2) isn’t the same as returning 2, for example.) You will get credit as long as these are either correct or consistent with a wrong answer to part 1. Although STk only accepts up to four as/ds between the c and r, you can put in as many as you want, but you should only be filling in as and ds.

(c________________________r honeyimsorry) returns 1

(c________________________r honeyimsorry) returns 2

(c________________________r honeyimsorry) returns 3
**Question 5: Poll Positions**

A certain polling place has a (positive) number of voting booths lined up in a single row. Voters are currently using some of them. However, any voter who is **directly** next to another voter (on either side or on both sides) feels *nervous*. (The voting booths on either end of the row are next to walls; walls do not make voters feel nervous.) Any voter who is not nervous is *calm*.

Write `(num-calm layout)`, **which takes a word layout (of positive length)** consisting only of `v`s (which represent voters using voting booths) and/or `e`s (which represent empty voting booths), and returns the number of calm voters. Examples:

```
STk> (num-calm 'v) 1
STk> (num-calm 'vev) 2
STk> (num-calm 'evve) 0
STk> (num-calm 'evvvev) 2
```
**Question 6: Tick Tock**

Ke$ha has just finished partying at the Telephone Nightclub, and she’s successfully gotten a little bit tipsy. Luckily, it’s a linear (one-dimensional) walk home from the club to her house. The club is at position 0, Ke$ha starts at position 1, and her house is at position $h$. ($h$ is 2 or greater.) Ke$ha can make one move per minute, in either direction (and **must** move; she can’t just stand still), but she only has $t$ minutes (and therefore $t$ moves) before she passes out.

If Ke$ha ever moves to position 0, where the club is, she immediately resumes partying (and makes no more moves), and therefore never gets home. If she ever moves to position $h$, she is successfully home (and makes no more moves). Even if she reaches home just as she runs out of time, it still counts as getting home. If she runs out of time and she is anywhere other than home, she obviously never gets home!

Write `(ways-home $t$ $h$)`, which returns the number of distinct ways in which Ke$ha can get home in $t$ minutes or less, given that her house is at position $h$. Here’s an example in which $h = 3$. (Remember that Ke$ha always starts at 1, and the club is always at 0.)

```
(ways-home 0 3) and (ways-home 1 3) should both return 0, since there’s no way to get home in time.
(ways-home 2 3) should return 1, since the only way to get home in time is to make two moves to the right. Even though she reaches home just as time runs out, it still counts as getting home.
(ways-home 3 3) should return 1, since Ke$ha can make it home with two moves to the right, as in the previous solution, but there is no other series of three moves or less that makes it home.
(ways-home 4 3) should return 2; the possibilities are two moves to the right, or four moves: right, left, right, right.
```
Space to answer Question 6
Question 7: Making the Grades

Imagine that Ian is storing the grades for CS3L in a big list called a “gradebook”. A “gradebook” is a list of “student”’s. A “student” is a two-item list:

* The first item is a word consisting of the last two letters of the student’s login.
* The second item is a list of that student’s scores on assignments.

Moreover, there is one fictional student, pp, whose list of “scores” represents the point values of the assignments. You can think of pp as standing for Points Possible. You may assume that any gradebook will always have at least one student besides pp and at least one assignment, all assignment values are positive, and every student has a score for every assignment. No logins will be duplicated, and students are stored in the gradebook in alphabetical order by login.

For example, here’s a simple gradebook that contains only two students, with logins da and db, plus pp, and data for two assignments worth 5 and 3 points, respectively. Student da got 4/5 on the first assignment and 2/3 on the second assignment.

(define my-grades '((da (4 2)) (db (2 2)) (pp (5 3))))

1. Complete the following (glookup login gb) procedure, which takes a student’s login and the name of a gradebook as an argument and returns that student’s overall grade (all points earned divided by all points possible, as a percentage out of 100). (glookup-helper takes a login and computes the total of all the scores recorded for that login.) Using the example above, (glookup 'da my-grades) should return 75. Fill in each blank with a single atom (i.e. not a compound expression), and do not otherwise add to / alter the existing code.

(define (glookup login gb)

  (define (glookup-helper who)
    (____________________)
    ______________________
    ______________________
    ______________________
    (____________________ (____________________ who gb)))
  (* (/ (glookup-helper login) (glookup-helper 'pp)) 100))
Question 7, continued

2. Write (add-assignment scores gb), which takes scores (which is a list of two-element lists of student logins and those students’ scores on the assignment, including pp’s “score”, which is the assignment’s point value) and a gradebook gb. add-assignment should return a new gradebook in which every student’s score has been added to the end of that student’s score data. The logins in scores are not necessarily in the same order as they are in gb. Moreover, not all logins in the gradebook will necessarily appear in scores; you should give any missing students a score of 0 for the assignment. However, you are guaranteed that all logins in scores are actually in the gb, and that no login has duplicate entries in scores. The returned gradebook must have the correct format for a gradebook, as described earlier.

For example: (add-assignment '((db 8) (pp 10)) my-grades) would return:

((da (4 2 0)) (db (2 2 8)) (pp (5 3 10)))
Question 8: Do as the Bromans Did

The little-known Broman civilization probably collapsed because of its number system. The Bromans used letters as digits, but they only had the letters B, O, and R. We know that B stood for 1, but we only know that O and R stood for different integer values o and r, both greater than 1. A Broman numeral’s value in our number system is equal to the sum of the values of its digits, with the exception that the sequence BRO (consecutively, in that order) is worth some value bro, and isn’t calculated as the sum of its parts 1 + r + o. So, if o = 5, r = 25, and bro = 100, then:

OR = 30; ROB = 31; BORO = 36; BOOR = 36; BROO = 105; BROBROBRO = 300

1. Write (from-broman numeral o r bro), which takes a Broman numeral and returns its decimal value, given that the values of O, R, and the consecutive BRO are o, r, and bro. Examples:

STk> (from-broman 'OR 33 66 100) 99
STk> (from-broman 'OBROB 3 30 100) 104
**Question 8, continued**

As you saw from the previous examples of BORO = 36 and BOOR = 36, the same number can be represented by more than one Broman numeral. But the Bromans had a *preferred* representation of each numeral: it was the possibility that was shortest, and, if one or more possibilities tied for shortest, it was the alphabetically earliest of the shortest possibilities. (Remember that (before? word1 word2) will return #t if word1 is alphabetically earlier than word2, and #f otherwise.)

2. Write (to-broman num o r bro), which takes a number num and returns the *preferred* Broman representation, given that the values of O, R, and the consecutive sequence BRO are o, r, and bro. Examples:

```
STk> (to-broman 7 5 10 6)  STk> (to-broman 12 3 6 10)
BBO          RR
STk> (to-broman 10 3 6 10)  STk> (to-broman 8 15 25 6)
BOR          BBBRO
```