CS3L Summer 2011 Exam 1
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 not use recursion in your solutions for this exam. This will only be a danger for people who have learned to use recursion, and you should know what not to do (don’t have your procedures call themselves directly or indirectly). Use of repeated is OK. If you have any doubts about whether something is “legal”, PLEASE ask us as soon as possible.

2. If you have difficulty understanding the language of a question or you’re not sure exactly what a 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>What Will Scheme Print?</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Master of Scheme Domains</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>No, Mr. Cond, I Expect You To Run... Forever!!!</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>The Silence of the Lambdas</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>That’s How We Roll</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Devoweluation</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Same Difference</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>The End</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TOTAL</td>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>
**Question 1: What Will Scheme Print?**

Suppose that you type each of the following into STk. **What will Scheme print in each case?** (Assume that you type these at separate times -- the subproblems don’t interact with each other -- and you haven’t previously defined any variables/procedures or otherwise changed Scheme from its default state.) If Scheme would print something like `[closure arglist]`, just write `procedure`. If you would get an error message (for any reason), just write `error`.

```
STk> (butlast '(butt last)) ______________________

STk> (every (remainder 12) '(1 2 3 4)) ______________________

STk> (repeated count 9876543210) ______________________

STk> (define x 1)
  x
STk> (x) ______________________

STk> (define (y) 2)
  y
STk> (y) ______________________

STk> (define (add-s wd)
       (word wd s))
  add-s
STk> (add-s 'fly) ______________________

STk> (define (concise login)
       (word (last (bl login))
            (last login)))
  concise
STk> (concise 'cs3-ok) ______________________

STk> (define (rank card)
       (let (((t 10) (j 11) (q 12)
               (k 13) (a 14))
              (last card)))
  rank
STk> (rank 'dq) ______________________
```
STk> (define (all-diff? x y z)
        (and (not (equal? x y))
             (not (equal? y z)))
all-diff?
STk> (all-diff? 'P 'G 'P)

STk> (define (enigma x y)
        (= y x)
        (+ x y)
        x)
enigma
STk> (enigma 1 2)

STk> ((lambda (expression)
        (let ((+ -) (- *))
          expression))
        (+ 5 4))

STk> (every (lambda (x)
            (every (lambda (y)
                 (- x y))
               '(1 2)))
         '(3 4))

Question 2: Master of Scheme Domains

a. Give the names of any two built-in Scheme procedures that satisfy the following characteristics:

Domain: must take exactly two arguments
Range: a Boolean value

b. Give the names of any three built-in Scheme procedures that must take exactly one argument, but are not predicate procedures. Here are some examples that you may not use, since they would make the question trivially easy: first, second, butfirst, last, butlast.
**Question 3: No, Mr. Cond, I Expect You To Run... Forever!!!**

A villain has captured the Scheme-savvy secret agent James Cond and forced him to write a procedure \texttt{r-f-third?} that takes three arguments. Cond knows that the villain will invoke the procedure with the three inputs \texttt{#t, #f, and (run-forever)}, in some order. \texttt{(run-forever)} is defined as usual: \texttt{(define (run-forever) (run-forever))}. The villain expects \texttt{r-f-third?} to return \texttt{#t} (without running forever) if and only if \texttt{(run-forever)} has been given as the third of the three inputs, but this may or may not be how Cond’s procedure will actually behave. Here’s what Cond has written:

\begin{verbatim}
(define (r-f-third? x y z)
  (and (not (and x y z))
       (or x y z)))
\end{verbatim}

**Circle** any and all of the following calls which will return \texttt{#t} (without running forever). **Cross out** any and all of the following calls which will not return \texttt{#t} (i.e., they will return \texttt{#f} or will run forever).

\begin{verbatim}
(r-f-third? #t #f (run-forever))    (r-f-third? #f #t (run-forever))
(r-f-third? #t (run-forever) #f)    (r-f-third? #f (run-forever) #t)
(r-f-third? (run-forever) #t #f)    (r-f-third? (run-forever) #f #t)
\end{verbatim}

**Question 4: The Silence of the Lambdas**

a. Rewrite the following code so that it is equivalent, but doesn’t contain any explicit lambdas, doesn’t use any helper functions, and includes exactly one \texttt{define} and exactly one \texttt{let}.

\begin{verbatim}
(define x (lambda () ((lambda (y z) (z y)) 1 - )))
\end{verbatim}
**Question 4, continued**

b. What will Scheme print when you enter x?  

______________________________

c. What will Scheme print when you enter (x)?  

______________________________

**Question 5: That's How We Roll**

The `threed6` procedure that you wrote in lab was inspired by a method for determining a score in the classic Dungeons & Dragons role-playing game. In more modern versions of D&D, the preferred method of generating a score is actually to **roll four six-sided dice, take away the lowest one of those dice, and then add the remaining three together**. For example, one might roll 3, 5, 5, and 6, and then drop the 3 and add the remaining dice to get 16. Write a procedure `fourd6droplowest`, with no arguments, that outputs the results of generating a score **in the exact method described in bold above**.

Hint: Remember that `(random 6)` returns a random number from the set {0, 1, 2, 3, 4, 5}.
**Question 6: Devoweluation**

Write a procedure `consonants` that takes a sentence `s` and returns a **single word** consisting only of all the consonants in `s` (in their original order). You may assume that `vowel?` is defined and has the usual behavior (takes a word, ideally one letter long, as an argument; returns `#t` if the input is `a`, `e`, `i`, `o`, or `u`, and `#f` otherwise).

Example:
STk> (consonants '(i can haz consonants))
cnhzcnsnnts
**Question 7: Same Difference**

For HW1, you probably wrote procedures like `all-three-same?` and `all-three-different?`; for HW2, you probably wrote procedures like `all-four-same?` and `all-four-different?`. But what if the makers of Set add more properties to their cards, or Lady Gaga decides to use larger hands with more suits? It’s a fundamental rule of software engineering that customers change their minds, so it would be wise to come up with more general solutions.

Write the procedures `all-same?` and `all-different?`, each of which can take a sentence of 2 or more words as its one argument. `all-same?` should return `#t` if all of its arguments are the same (in the sense of `equal?`) and `#f` otherwise. `all-different?` should return `#t` if all of its arguments are different (in the sense of `equal?`) and `#f` otherwise. Assume that the words in these sentences will always be single characters like letters or one-digit numbers.

Examples:
STk> (all-different? ' (c d h s t 7 !))
#t
STk> (all-same? ' (5 5 5 5 6 5))
#f

You may find `all-same?` easier to write than `all-different?`. 
**Question 8: The End**

We often wish to search for words that end with a certain series of characters, e.g., "Find all the files in this UNIX directory that end with .scm".

Write a procedure *ending-matches* that takes two arguments: a word *ending* (which may be of any length, including the empty word) and a sentence *s* (containing words that may be of any length). *ending-matches* should return a new sentence consisting of all the words in *s* that end exactly with *ending* (that is, they consist of zero or more characters followed by the exact series of characters in *ending*, with nothing thereafter).

Examples:
STk> (ending-matches ' .scm ' (core hw lab1.scml lab1.scml scm))
(lab1.scml)
STk> (ending-matches ' one ' (one ones 1 done phone money))
(one done phone)