Read and fill in this page now

Your name: ___________________________________

Your login name: ___________________________________

Your lab section day and time: ___________________________________

Your lab T.A.: ___________________________________

Name of the person sitting to your left: ___________________________________

Name of the person sitting to your right: ___________________________________

Prob 0: _______  Prob 2: _______  Prob 4: _______

Prob 1: _______  Prob 3: _______  Prob 5: _______

________  __________

Total ___________/28

You have one hour to finish this test, which should be reasonable; there will be approximately 20 minutes of leeway given past one hour. Your exam should contain 6 problems (numbered 0-5) on 8 pages.

This is an open-book test. You may consult any books, notes, or other paper-based inanimate objects available to you. Read the problems carefully. If you find it hard to understand a problem, ask us to explain it. If you have a question during the test, please come to the front or the side of the room to ask it.

Restrict yourself to Scheme constructs covered in chapters 3-14 of Simply Scheme, all parts of the “Difference Between Dates” case study, the “Roman Numerals” case study, and all lab materials.

Please write your answers in the spaces provided in the test; if you need to use the back of a page make sure to clearly tell us so on the front of the page. We believe we have provided more than enough space for your answers, so please don’t try to fill it all up.

Partial credit will be awarded where we can, so do try to answer each question.

Relax!
Problem 0. Your name, please! (1 point, 1 minute)

Put your login name on each page. Also make sure you have provided the information requested on the first page.

Problem 1. Remove consecutive duplicates (6 points, 11 minutes)

Consider a function `remove-conseq-dups` that takes a sentence and returns a sentence in which any occurrences of a word that follow that same word are removed. Note that the sentence may be of any length (e.g., empty).

For instance,

| `(remove-conseq-dups '(the rain um um in spain um falls um um um mainly on on on um the plain))` | `(the rain um in spain um falls um mainly on um the plain)` |
| `(remove-conseq-dups '(a b a b c b))` | `(a b a b c b)` |

**Part A:**
Write `remove-conseq-dups` without using higher order functions (i.e., using recursion). You may use helper procedures.
Part B:
Write remove-conseq-dups without using explicit recursion (i.e., using higher order functions). You may use helper procedures.
Problem 2. **Rewrite** `process-it` **as a HOF (4 points, 8 minutes)**

Rewrite `process-it` below using higher order functions (with no explicit recursion).

```scheme
(define (process-it wd sent pred?)
  (cond ((empty? sent) `())
        ((pred? (first sent))
         (se (word wd (first sent))
             (process-it wd (bf sent) pred?)))
        (else
         (process-it wd (bf sent) pred?))))
```
Problem 3. Growing buggy mountains (6 points, 11 minutes)

Consider a procedure mountains which takes a word as input, and returns a sentence containing words starting as the first letter of the input, then grow up to the full input word, and then back to the first letter.

\[
\text{(mountains 'cs3)} \rightarrow \text{(c cs cs3 cs c)}
\]

Part A
The table below contains versions of mountains that novice students submitted. (Not this brilliant class, of course, but some alternate universe). For each version, describe what the call \(\text{(mountains 'cs3)}\) will return in the box below. If a crash will occur, indicate why and where it happens, as specifically as possible. If the procedure will run infinitely, note that.

\[
\begin{array}{|l|}
\hline
\text{(define (mountains wd)} \\
\quad (if (empty? wd) '()) \\
\quad \quad (se (mountains (bl wd))) \\
\quad \quad wd \\
\quad \quad (mountains (bl wd)))) \\
\hline
\end{array}
\]

\[
\begin{array}{|l|}
\hline
\text{(define (mountains wd)} \\
\quad (se (mountains-helper wd) \\
\quad \quad wd \\
\quad \quad (reverse (mountains-helper wd)))) \\
\hline
\end{array}
\]

\[
\begin{array}{|l|}
\hline
\text{(define (mountains-helper wd)} \\
\quad (if (empty? wd) '()) \\
\quad \quad (se (mountains-helper (bl wd)) wd)) \\
\hline
\end{array}
\]
Part B
Someone decides that mountains would look better if the sentence ended in the last letter of the input word, rather than the first:

\[
\begin{array}{|l|}
\hline
\text{(better-mountains 'cs3)} & \rightarrow & \text{(c cs cs3 s3 3)} \\
\hline
\text{(better-mountains 'fred)} & \rightarrow & \text{(f fr fre fred red ed d)} \\
\hline
\end{array}
\]

In fact, it does look better.

Write better-mountains.
Problem 4. Fill in the blanks (6 points, 10 minutes)

Fill in the blank to define a procedure twice. As input, twice takes a function \( f \) that takes a single input. As output, twice returns a procedure which takes a single input and applies \( f \) twice.

For instance, if twice were given a function that adds 3 to its input, twice would return a function that adds 6 to its input. If twice were given the function last, it would return a function that, when given a sentence, would return everything but the last two elements of the sentence.

\[
\text{(define (make-twice func)}
\]

\[
\text{(___________________________________))}
\]

\[
\text{(every (lambda (x)}
\]

\[
\text{(cond ((negative? x) (* x -1))}
\]

\[
\text{((zero? x) 'zero)}
\]

\[
\text{(else '('))))}
\]

\[
\text{') (2 6 -4 4 0 -9 0 9))}
\]

\[
\text{⇒ ____________________________________________________________)}
\]

Fill in the blank to define a procedure make-member, which returns a procedure that takes a word and returns true only if the word is in the sentence passed to make-member.

\[
\text{(define (make-member sent)}
\]

\[
\text{(__________________________________________________________)}
\]
Problem 5. Modify the pascal program (6 points, 10 minutes)

The pascal procedure calculates the number in the corresponding cell of pascal's triangle, given valid row and column values:

\[
\text{(define (pascal col row)}
\text{ (cond ((equal? col 0) 1)}
\text{ ((equal? col row) 1)}
\text{ (else (+ (pascal col (- row 1))
\text{ (pascal (- col 1) (- row 1))))))})
\]

<table>
<thead>
<tr>
<th>rows (R)</th>
<th>columns (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 2 3 4 5 ...</td>
</tr>
<tr>
<td>1</td>
<td>1 1 ...</td>
</tr>
<tr>
<td>2</td>
<td>1 2 1 ...</td>
</tr>
<tr>
<td>3</td>
<td>1 3 3 1 ...</td>
</tr>
<tr>
<td>4</td>
<td>1 4 6 4 1 ...</td>
</tr>
<tr>
<td>5</td>
<td>1 5 10 10 5 1 ...</td>
</tr>
<tr>
<td>...</td>
<td>... ... ... ... ... ...</td>
</tr>
</tbody>
</table>

Write a procedure pascal-calls that counts the number of recursive invocations that a particular call to pascal makes – that is, how many times pascal calls itself. pascal-calls will need to take a column and a row as input.

Provide three good test-cases (with return values) for valid invocations to pascal-calls.