**CSSE 304 Exam #1 Part 2 Sep 24, 2019 (day 14) Name\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Section: 01 (9:00) 02 (10:00) 03 (11:00)**

|  |  |  |  |
| --- | --- | --- | --- |
| Problem | Possible | Earned | Grader Comments |
| C1 | 10 |  |  |
| C2 | 10 |  |  |
| C3 | 10 |  |  |
| C4 | 10 |  |  |
| Total | 40 |  |  |

During the entire exam you may not use email, IM, cell phone, PDA, MP3 player, headphones, ear buds, or any other communication device or software. Efficiency and elegance will not affect your score **on problems C1-C3**, provided that I can understand your code.

You may assume that all of your procedures' input arguments have the correct types and values; your code does not need to check for illegal input data. **Mutation is not allowed, except for the last problem, which requires mutation.**

**Part 2, programming.** You may use your notes, *Chez* Scheme, the three textbooks from the course, The *Chez* Scheme Users' Guide, the PLC grading program, any materials that I provided online for the course. You may do searches for built-in Scheme procedures, but not for the particular problems that you are solving. You may not use any other web or network resources, or programs written by other (past or present) students. You are allowed to look at and use any Scheme code that *you* have previously written.

**The Problems:**

**The first problem requires you to use Scheme string procedures.** I recognize that you have seldom or never used them, and looking those procedures up in TSPL is part of the point of these problems. There are a lot of built-in string procedures; to help you not be overwhelmed, I list ones that I used to solve these problems, plus some others. You may use other string procedures also.

string-length string-=? string-ref make-string substring string-append

**C1. (10 points)** (string-index ch str) takes a character ch and a string str as its arguments. It returns the zero-based position within str of the first occurrence of ch. If ch does not occur in str, string-index returns -1.

(string-index #\h "ether") 🡺 2

(string-index #\e "ether") 🡺 0

(string-index #\m "ether") 🡺 -1

**C2. (10 points)** (merge-2-sorted-lists lon1 lon2) takes two sorted lists of numbers and returns one sorted list that contains all of those numbers. Duplicates are preserved in the resulting list. At most half credit if your code doesn’t run in linear time. You are not allowed to use *Chez* Scheme's merge or merge! procedures.

> (merge-2-sorted-lists '() '(1 3))

(1 3)

> (merge-2-sorted-lists '(1 4 5 7 8) '(2 3 6))

(1 2 3 4 5 6 7 8)

> (merge-2-sorted-lists '(5 6) '(3 4))

(3 4 5 6)

> (merge-2-sorted-lists '(1 3 5) '(2 4 6))

(1 2 3 4 5 6)

> (merge-2-sorted-lists '(1 2 4 6) '(1 3 5 6))

(1 1 2 3 4 5 6 6)

**C3. (10 points)** (slist-equal? s1 s2) takes two s-lists as arguments and determines whether they are equal. You are not allowed to call equal? or any other built-in procedure that is not constant-time. As in some other slist problems:

1. Must be O(N) where N is the total number of parentheses in s1 and s2.

2. You can't traverse any sublist of either s-list twice.

3. You must work directly with s1, and s2; you may not transform them into a different data structure (such as a flat list).

4. It must short-circuit. Once you have found a place where the slists are different, you must immediately return #f without visiting the rest of the list.

**Hint:** Let the structure of your code be based on recursion over s1.

As a reminder, <s-list> ::= ( {<s-exp>}\* )

<s-exp> ::= <symbol> | <s-list>

> (slist-equal? '() '(a))

#f

> (slist-equal? '(a) '(b))

#f

> (slist-equal? '(a) '(a))

#t

> (slist-equal? '(a) '(() a))

#f

> (slist-equal? '((a)) '(a))

#f

> (slist-equal? '((a ((b) (c) ())) d) '((a ((b) (c) ())) d))

#t

> (slist-equal? '((a ((b) (c) ())) d) '((a ((b) (e) ())) d))

#f

> (slist-equal? '((a ((b) (c e) ())) d) '((a ((b) (c) ())) d))

#f

> (slist-equal? '((a ((b) (c) ())) d) '((a ((b) (c) ())) f))

#f

**C4. (10 points)** (make-queue)creates an initially-empty queue “object”, with the expected methods, enqueue, dequeue, empty?. The transcript below illustrates how these work. You probably want to store the queue as a list, with the car of the list being the front of the queue. In order to make all three queue operations be constant time, your queue “class” should probably have two instance fields, first and last. When the queue is empty, first should probably be null.

> (let ([q1 (make-queue)] [q2 (make-queue)])

(q1 'enqueue 1)

(q1 'dequeue))

1

> (let ([q1 (make-queue)] [q2 (make-queue)])

(q1 'enqueue 1)

(q2 'enqueue 2)

(q1 'dequeue)

(list (q1 'empty?) (q2 'empty?)))

(#t #f)

> (let ([q1 (make-queue)] [q2 (make-queue)])

(q1 'enqueue 1)

(q2 'enqueue 2)

(list (q1 'dequeue) (q2 'dequeue)))

(1 2)

> (let ([q1 (make-queue)] [q2 (make-queue)])

(q1 'enqueue 1)

(q2 'enqueue 2)

(q1 'enqueue (q2 'dequeue))

(let ([val (q1 'dequeue)])

(cons val (list (q1 'dequeue)))))

(1 2)

> (let ([q1 (make-queue)] [q2 (make-queue)])

(q1 'enqueue 1)

(q2 'enqueue 2)

(q1 'enqueue (q2 'dequeue))

(q2 'enqueue 3)

(list (q2 'dequeue) (q1 'dequeue)))

(3 1)