**CSSE 304 Exam #2 Part 2 Jan 8, 2020 (day 14.5) Name\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Section 01 (10:00) 02 (11:00) 03 (12:00)**

|  |  |  |  |
| --- | --- | --- | --- |
| Problem | Possible | Earned | Grader Comments |
| C1  prefix-sums | 6 |  |  |
| C2  suffix-sums | 6 |  |  |
| C3  evens-odds | 8 |  |  |
| C4  notate-depth-and-flatten | 10 |  |  |
| C5  free-occurrence-count | 10 |  |  |
| C6  curry | 10 |  |  |
| Total | 50 |  |  |

You may not use email, IM, cell phone, PDA, MP3 player, headphones, ear buds, or any other communication device or software.

**Part 2, programming.** You may use your notes, *Chez* Scheme, the three textbooks from the course, The *Chez* Scheme Users' Guide, 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 use any code that *you* have previously written. 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.**

Efficiency and elegance will not affect your score **on problems C4-C6**, provided that I can understand your code.   
Be careful not to use so much time on one problem that you do not get to work on other problems.

**C1. (6 points)** (prefix-sums lon) takes list of numbers lon as its argument. It returns a list of the sums of the non-empty prefixes of lon. Example: for the list (1 3 5 2) the prefix sums list is (1 1+3 1+3+5 1+3+5+2) which is (1 4 9 11). **For full credit, your code must run in O(N) time.**

> (**prefix-sums '(1 2 3 4 5 6))**

(1 3 6 10 15 21)

> **(prefix-sums '(1 3 5 2))**

(1 4 9 11)

> (prefix-sums '(4))

(4)

> **(prefix-sums '(3 2 0 -5 6))**

(3 5 5 0 6)

**C2. (6 points)** (suffix-sums lon) is similar to the above procedure, but it returns a list of the sums of the suffixes.

**For full credit, your code must run in O(N) time.**

> **(suffix-sums '(1 2 3 4 5 6))**

(21 20 18 15 11 6)

> **(suffix-sums '(1 3 5 2))**

(11 10 7 2)

> **(suffix-sums '(4))**

(4)

> **(suffix-sums '(3 2 0 -5 6))**

(6 3 1 1 6) (3 4 5 6)

**C3. (10 points)** (evens-odds ls) takes a list ls as its argument. It returns a list of two lists: the elements in even positions within ls and the elements in odd positions in ls. Both of the returned lists should be in the same order as those elements occur within **ls**. **For full credit, your code must run in O(N) time, and it must only traverse the original list ls once.**

> **(evens-odds '())**

(() ())

> **(evens-odds '(b))**

((b) ())

> **(evens-odds '(c d))**

((c) (d))

> **(evens-odds '(a b c d e f g))**

((a c e g) (b d f))

> **(evens-odds '(b c d e f g))**

((b d f) (c e g))  
  
  
**C4. (10 points)** (notate-depth-and-flatten slist) takes an s-list slist as its arguments. It returns a flat version of slist, with each symbol annotated to indicate its depth within the original s-list. You may use notate-depth as a helper procedure.

> **(notate-depth-and-flatten '())**

()

> **(notate-depth-and-flatten '(a ((b))))**

((a 1) (b 3))

> **(notate-depth-and-flatten '((a () (b c (() d))) e))**

((a 2) (b 3) (c 3) (d 4) (e 1))

**C5. (10 points)** (free-occurrence-count exp) takes an LCexp exp as its argument. It returns the number of free occurrences of all variables in exp. Use the original definition of LCexp, not the one that you enhanced for the later problems in A10. You can find the grammar and the definition of occurs-free on pages 6 and 7 of the Session 11 slides.

> **(free-occurrence-count '(x (x y)))**

3

> **(free-occurrence-count**

**'(lambda (x) (lambda (x) (x y))))**

1

> **(free-occurrence-count**

**'(lambda (x) (lambda (y) (x (x (y y))))))**

0

> **(free-occurrence-count**

**'(lambda (x) (lambda (w) (x ((x z) (y ((x y) (y w))))))))**

4

> **(free-occurrence-count**

**'((lambda (x) (y (lambda (y) (x y))))**

**(lambda (y) ((x x) (lambda (x) ((z z) (y (x x))))))))**

5

**C6. (10 points)** (curry n f) takes as its arguments a positive integer n and a procedure f that can take n arguments. It returns a procedure that is a completely curried version of f. This is a generalization of the curry2 procedure that you wrote in the homework. In fact, if f is a procedure that can take two arguments, (curry 2 f) does the same thing as (curry2 f). It is possible to pass these test cases by writing special cases for n = 1,2, 3, 7, and 12. You will earn no points if you do that. For full credit, your code should work for any positive n.

> **(let ([+-c (curry 3 +)])**

**(((+-c 2) 4) 7))**

13

> **(let ([cons-c (curry 2 cons)])**

**((cons-c 4) '(5 6)))**

(4 5 6)

> **(let ([car-c (curry 1 car)])**

**(car-c '(1 2 3)))**

1

> **(let ([+-c (curry 7 +)])**

**(((((((+-c 2) 4) 6) 8) 10) 12) 14))**

56

> **(let ([+-c (curry 12 +)])**

**((((((((((((+-c 1) 2) 3) 4) 5) 6) 7) 8) 9) 10) 11) 12))**

78