**CSSE 304 Final Exam Computer part November 18, 2020**

You may assume that for any procedure you are asked to write, all input arguments will be of the correct types; you do not need to

write code to check for illegal input data.

For this part, you may use resources that I have placed online, plus TSPL and EoPL, your notes, and a Scheme programming environment, plus the PLC grading server. You are allowed to look at and use any Scheme code on your computer that **you** have previously written.

The test cases in the grading program will be the examples given in the problem descriptions, so there should be no surprises. If the grading program gives you all of the points for a problem and you did not “program to the test cases”, that will be your score. If not, we will look at your code, and we may give you a different amount of partial credit than the grading program gives you, based on how much understanding your code reflects. **Caution!** It is possible to get so caught up in getting all of the points for one computer problem that you do not get to the other problems. Don’t do that! We will give partial credit if you have the main ideas, even if a procedure does not produce correct answers for any test cases.

Your submitted code should **not** include any output-producing statements, such as display, write, or trace.

**1. (AA points)**  IF YOU WANT TO HAVE TIME TO DO ALL OF THE PROBLEMS, YOU SHOULD PROBABLY SPEND NO MORE THAN 30 MINUTES ON THIS ONE. Write a procedure **permute**, that takes a nonempty sorted list of numbers with no duplicates and returns a list of all permutations of that list, in "increasing order" as in the examples below. You are allowed (but not required) to get **permute** working for an empty list also, if it is easier for you to write the code that way (It was easier for me to write the code that way). You do not have to test for correct input format; you may assume it. For full credit, you must not use mutation. You may earn up to ½ of the points if you use mutation in your solution. **Hint:** I found remove (built-in) to be helpful.

> **(permute '())**

()

> **(permute ' (1))**

((1))

> **(permute '(2 4 6))**

((2 4 6) (2 6 4) (4 2 6) (4 6 2) (6 2 4) (6 4 2))

> **(permute '(3 5 7 9))**

((3 5 7 9) (3 5 9 7) (3 7 5 9) (3 7 9 5) (3 9 5 7) (3 9 7 5) (5 3 7 9) (5 3 9 7) (5 7 3 9)

(5 7 9 3) (5 9 3 7) (5 9 7 3) (7 3 5 9) (7 3 9 5) (7 5 3 9) (7 5 9 3) (7 9 3 5) (7 9 5 3)

(9 3 5 7) (9 3 7 5) (9 5 3 7) (9 5 7 3) (9 7 3 5) (9 7 5 3))  
> **(permute '(0 2 4 6 8))**

((0 2 4 6 8) (0 2 4 8 6) (0 2 6 4 8) (0 2 6 8 4) (0 2 8 4 6)

(0 2 8 6 4) (0 4 2 6 8) (0 4 2 8 6) (0 4 6 2 8) (0 4 6 8 2)

(0 4 8 2 6) (0 4 8 6 2) (0 6 2 4 8) (0 6 2 8 4) (0 6 4 2 8)

(0 6 4 8 2) (0 6 8 2 4) (0 6 8 4 2) (0 8 2 4 6) (0 8 2 6 4)

(0 8 4 2 6) (0 8 4 6 2) (0 8 6 2 4) (0 8 6 4 2) (2 0 4 6 8)

(2 0 4 8 6) (2 0 6 4 8) (2 0 6 8 4) (2 0 8 4 6) (2 0 8 6 4)

(2 4 0 6 8) (2 4 0 8 6) (2 4 6 0 8) (2 4 6 8 0) (2 4 8 0 6)

(2 4 8 6 0) (2 6 0 4 8) (2 6 0 8 4) (2 6 4 0 8) (2 6 4 8 0)

(2 6 8 0 4) (2 6 8 4 0) (2 8 0 4 6) (2 8 0 6 4) (2 8 4 0 6)

(2 8 4 6 0) (2 8 6 0 4) (2 8 6 4 0) (4 0 2 6 8) (4 0 2 8 6)

(4 0 6 2 8) (4 0 6 8 2) (4 0 8 2 6) (4 0 8 6 2) (4 2 0 6 8)

(4 2 0 8 6) (4 2 6 0 8) (4 2 6 8 0) (4 2 8 0 6) (4 2 8 6 0)

(4 6 0 2 8) (4 6 0 8 2) (4 6 2 0 8) (4 6 2 8 0) (4 6 8 0 2)

(4 6 8 2 0) (4 8 0 2 6) (4 8 0 6 2) (4 8 2 0 6) (4 8 2 6 0)

(4 8 6 0 2) (4 8 6 2 0) (6 0 2 4 8) (6 0 2 8 4) (6 0 4 2 8)

(6 0 4 8 2) (6 0 8 2 4) (6 0 8 4 2) (6 2 0 4 8) (6 2 0 8 4)

(6 2 4 0 8) (6 2 4 8 0) (6 2 8 0 4) (6 2 8 4 0) (6 4 0 2 8)

(6 4 0 8 2) (6 4 2 0 8) (6 4 2 8 0) (6 4 8 0 2) (6 4 8 2 0)

(6 8 0 2 4) (6 8 0 4 2) (6 8 2 0 4) (6 8 2 4 0) (6 8 4 0 2)

(6 8 4 2 0) (8 0 2 4 6) (8 0 2 6 4) (8 0 4 2 6) (8 0 4 6 2)

(8 0 6 2 4) (8 0 6 4 2) (8 2 0 4 6) (8 2 0 6 4) (8 2 4 0 6)

(8 2 4 6 0) (8 2 6 0 4) (8 2 6 4 0) (8 4 0 2 6) (8 4 0 6 2)

(8 4 2 0 6) (8 4 2 6 0) (8 4 6 0 2) (8 4 6 2 0) (8 6 0 2 4)

(8 6 0 4 2) (8 6 2 0 4) (8 6 2 4 0) (8 6 4 0 2) (8 6 4 2 0))

**Submit your code to the F-202110-1 assignment on the PLC server.**

**2. (BB points)** IF YOU WANT TO HAVE TIME TO DO ALL OF THE PROBLEMS, YOU SHOULD PROBABLY SPEND NO MORE THAN 10 MINUTES ON THIS ONE. We have used procedures with persistent variables to simulate OOP in Scheme, but we have not had an approach that allows an “object” to refer to itself (like this in Java or self in Python). I have provided a modified version of the code for my make-queue constructor from Exam 2. This code does not work because self is not defined. Simple changes in the code will fix this. Do not make any changes to any of the cases in the case code. For full credit, your new code must not create or modify any global variables or procedures.

(define make-queue ; fix this, but do not change anything in the case.

(lambda ()

(let ([front '()] [rear '()])

(lambda args

(case (car args)

[(enqueue!)

(let ([x (list (cadr args))])

(if (null? front)

(set! front x)

(set-cdr! rear x))

(set! rear x))]

[(enqueue-from-list!)

(let loop ([ls (cadr args)])

(if (not (null? ls))

(begin

(self 'enqueue! (car ls))

(loop (cdr ls)))))]

[(empty?) (null? front)]

[(display) (printf "contents: ~s ~s~%" front rear)] ; if needed for debugging.

[(dequeue!) (let ([obj (car front)])

(set! front (cdr front))

(if (null? front)

(set! rear '()))

obj)])))))

**Test case:**

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

(q1 'enqueue! 5)

(q1 'enqueue! 6)

(q1 'enqueue-from-list! '(1 2 3 4))

(q2 'enqueue-from-list! '(5 6 1))

(q2 'enqueue! 2)

(q2 'enqueue! 3)

(q2 'enqueue! 4)

(let loop ()

(cond [(q1 'empty?) #t]

[(= (q1 'dequeue!)(q2 'dequeue!)) (loop)]

[else #f])))

**Submit your code to the F-202110-2 assignment on the PLC server.**

**3. (CC points)** Below (and in the starting code) are the definitions of slist-reverse (a substantial procedure) and snoc (treat   
 it as a primitive procedure). Test cases are in the starting code file. Feel free to unquote them when you are ready to run them.

(a) rewrite slist-reverse as slist-reverse-cps with Scheme-procedure continuations. Use make-k and   
 apply-k to indicate where you are creating and using continuations.

(b) transform to data-structures continuations, using the datatype that I have begun for you, and renaming your substantial   
 procedures to slist-reverse-ds and apply-k-ds.

(c ) transform to imperative form, renaming your substantial procedures to slist-reverse-imp and apply-k-imp.   
 Add any global variables that are needed.

**Submit your code to the F-202110-3 assignment on the PLC server.**

**Interpreter problems strategy.** You only have to do one of the two interpreter problems, and we will only grade one of them (the one for which the PLC server gives you the most points). You have potential to earn all YY points if you do problem 5. But some students will not have the level of understanding necessary to do problem 5. Therefore I have provided problem 4 as a simpler alternative (worth only XX points) that most D+ or C students should be able to do in the available time.

My suggestion is that you spend a few minutes thinking about problem 5. Write down which parts of the code need to be changed and a very brief description of what needs to happen jn each of those places. If you cannot do that with a reasonably high degree of confidence that your plan will work, or if after doing it you have fewer than 45 minutes left for coding and debugging, I suggest that you do problem 4 as a “safety net” so you can at least get those XX points for this YY-point problem. Then if you have time left, you may want to go back to work on problem 5.

**4. (XX points)** Some languages (Python, for example) include multiple assignment statements, such as a,b,c = 1,2,3. This example is just convenient shorthand, but multiple assignment can make code simpler in some cases. For example, to exchange the values assigned to a and b, we can write a,b = b,a. The semantics are that the expressions on the right-hand side are all evaluated (using the original values of the left-hand-side variables), and then the results from the right-hand side are assigned to the corresponding variables on the left-hand side. A convenient Scheme-like notation is

(mset! (id1 id2 … idn) (exp1 exp2 … expn)) ,

where n > 0, and the number of expressions must equal the number of identifiers. Like its cousin set!, mset! does not create new bindings; all of the ids must already be bound. For simplicity for this timed exam, you may assume that none of the expressions in (exp1 exp2 … expn) cause side-effects.

For example,

-->(let ([a 1] [b 2] [c 3])

(mset! (a b c) ((+ b c) (+ 1 c) (+ a (+ b c))))

(list a b c))

(5 4 6)

-->(let ([a 2] [b 3])

(mset! (a b) (b a))

(list a b))

(3 2)

**5. (YY points)** This can be done starting with an A17a interpreter and applying call/cc to get the needed continuations, or starting with an A18a or A18b interpreter and using the continuations that the interpreter already knows about. The A18 approach is probably easier of you have a working A18 interpreter Either way, a global list of continuations (treated as a stack) may be helpful.

Add a break statement to the language recognized by your interpreter. This does not behave like Scheme's break, but is more like a dynamic version of Java's break.

<expression> ::= (break <expression>) *; expression must evaluate to a non-negative integer*

<expression> ::= (break)

If (break) or, equivalently, (break 0) is encountered during the execution of the body of a while loop (including applications of procedures called by the body of that while loop), that loop immediately terminates, and the code that would normally have executed after the while loop finished will now execute. In general, (break n) immediately exits from *n*+1 levels of while loops. If (break n) is used in an execution context where there are not *n*+1 while loops "in progress", your interpreter could print an error message and end execution of the current expression, but I will not test that illegal use of break.

--> **(let ([a 6] [sum 0]) ; not an official test case.**

**(while (> a 0)**

**(begin**

**(set! sum (+ sum a))**

**(if (= a 3) (break))**

**(set! a (- a 1))))**

**sum)**

18

While a lexical break might be easier for the user, a dynamic break is easier to implement, so that's what you'll do:

-->**(let ([helper ; This is also not an “official” test case.**

**(lambda (t)**

**(if (= t 3)**

**(break)**

**(\* t 2)))]**

**[a 7]**

**[sum 0])**

**(while (> a 0)**

**(set! sum (+ sum (helper a)))**

**(if (= a 3) (break))**

**(set! a (- a 1)))**

**sum)**

44

**Actual tests:**

> **(eval-one-exp**

**'(let ([sum 0] [max 6])**

**(while (> max 0)**

**(begin (set! sum (+ sum max))**

**(set! max (- max 1))**

**(if (= max 3) (break))))**

**sum))**

15

> **(eval-one-exp**

**'(let ([i 0] [sum 0])**

**(while (< i 6)**

**(begin (if (= i 3)**

**(break))**

**(set! i (+ i 1))**

**(set! sum (+ i sum))))**

**(list i sum)))**

(3 6)

> (**eval-one-exp**

**'(let ([sum 0]**

**[max 6]**

**[test (lambda (n) (if (= n 3) (break)))])**

**(while (> max 0)**

**(begin (set! sum (+ sum max))**

**(set! max (- max 1))**

**(test max)))**

**sum))**

15

> **(eval-one-exp**

**'(let ([main-list (list)] [max 5])**

**(while (> max 0)**

**(let ([counter max][inner-list (list)])**

**(while (> counter 0)**

**(begin**

**(set! inner-list (cons counter inner-list))**

**(if (= counter 3)**

**(break))**

**(set! counter (- counter 1))))**

**(set! main-list (cons inner-list main-list))**

**(set! max (- max 1))))**

**main-list))**

((1) (1 2) (3) (3 4) (3 4 5))

> **(eval-one-exp**

**'(let ([main-list '()] [max 1])**

**(while (<= max 6)**

**(let ([counter 1][inner-list '()])**

**(while (<= counter max)**

**(begin**

**(set! inner-list (cons counter inner-list))**

**(if (= counter 4)**

**(break 1))**

**(set! counter (+ counter 1))))**

**(set! main-list (cons inner-list main-list))**

**(set! max (+ max 1))))**

**main-list))**

((3 2 1) (2 1) (1))

> **(eval-one-exp**

**'(let ([i 1] [j 1] [k 1] [m 1] [sum 0])**

**(while (< i 3)**

**(begin**

**(set! i (+ i 1))**

**(set! j 1)**

**(while (<= j i)**

**(begin**

**(set! j (+ 1 j))**

**(set! k 1)**

**(while (<= k (\* j j))**

**(begin**

**(set! k (+ k i))**

**(set! m 1)**

**(while (<= m k)**

**(begin**

**(set! sum (+ sum 1))**

**(if (> sum 20) (break))**

**(set! m (+ m 1)))**

**)))))))**

**(list i j k m sum)))**

(3 4 19 1 34)

> **(eval-one-exp**

**'(let ([i 1] [j 1] [k 1] [m 1] [sum 0])**

**(while (< i 3)**

**(begin**

**(set! i (+ i 1))**

**(set! j 1)**

**(while (<= j i)**

**(begin**

**(set! j (+ 1 j))**

**(set! k 1)**

**(while (<= k (\* j j))**

**(begin**

**(set! k (+ k i))**

**(set! m 1)**

**(while (<= m k)**

**(begin**

**(set! sum (+ sum 1))**

**(if (> sum 20) (break 0))**

**(set! m (+ m 1)))**

**)))))))**

**(list i j k m sum)))**

(3 4 19 1 34)

> **(eval-one-exp**

**'(let ([i 1] [j 1] [k 1] [m 1] [sum 0])**

**(while (< i 3)**

**(begin**

**(set! i (+ i 1))**

**(set! j 1)**

**(while (<= j i)**

**(begin**

**(set! j (+ 1 j))**

**(set! k 1)**

**(while (<= k (\* j j))**

**(begin**

**(set! k (+ k i))**

**(set! m 1)**

**(while (<= m k)**

**(begin**

**(set! sum (+ sum 1))**

**(if (> sum 20) (break 1))**

**(set! m (+ m 1)))**

**)))))))**

**(list i j k m sum)))**

(3 4 4 1 24)

> **(eval-one-exp**

**'(let ([i 1] [j 1] [k 1] [m 1] [sum 0])**

**(while (< i 3)**

**(begin**

**(set! i (+ i 1))**

**(set! j 1)**

**(while (<= j i)**

**(begin**

**(set! j (+ 1 j))**

**(set! k 1)**

**(while (<= k (\* j j))**

**(begin**

**(set! k (+ k i))**

**(set! m 1)**

**(while (<= m k)**

**(begin**

**(set! sum (+ sum 1))**

**(if (> sum 20) (break 2))**

**(set! m (+ m 1)))**

**)))))))**

**(list i j k m sum)))**

(3 2 4 1 22)

> **(eval-one-exp**

**'(let ([i 1] [j 1] [k 1] [m 1] [sum 0])**

**(while (< i 3)**

**(begin**

**(set! i (+ i 1))**

**(set! j 1)**

**(while (<= j i)**

**(begin**

**(set! j (+ 1 j))**

**(set! k 1)**

**(while (<= k (\* j j))**

**(begin**

**(set! k (+ k i))**

**(set! m 1)**

**(while (<= m k)**

**(begin**

**(set! sum (+ sum 1))**

**(if (> sum 20) (break 3))**

**(set! m (+ m 1)))**

**)))))))**

**(list i j k m sum)))**

(2 3 7 5 21)

> **(eval-one-exp**

**'(let\* ([i 1]**

**[k 1]**

**[sum 0]**

**[break-test (lambda (num) (break (- 1 num)))])**

**(while (< i 6)**

**(begin**

**(set! k 1)**

**(while (<= k i)**

**(begin**

**(set! sum (+ sum i))**

**(if (= sum 8) (break-test 0))**

**(set! k (+ 1 k))))**

**(set! i (+ 1 i))))**

**(list i k sum)))**

(3 1 8)

> **(eval-one-exp**

**'(let\* ([i 1]**

**[k 1]**

**[sum 0]**

**[break-test (lambda (num) (break (- 1 num)))])**

**(while (< i 6)**

**(begin**

**(set! k 1)**

**(while (<= k i)**

**(begin**

**(set! sum (+ sum i))**

**(if (= sum 8) (break-test 1))**

**(set! k (+ 1 k))))**

**(set! i (+ 1 i))))**

**(list i k sum)))**

(6 6 49)

> **(let ([i 1] [j 1] [k 1] [sum 0])**

**(while (<= i 3)**

**(begin (set! j 1)**

**(while (<= j i)**

**(begin**

**(if (= j 2) (break 0))**

**(set! sum (+ sum i j))**

**(set! j (+ 1 j))))**

**(set! k 1)**

**(while (<= k i)**

**(begin**

**(set! sum (+ sum k))**

**(set! k (+ 1 k))))**

**(set! i (+ 1 i))))**

**(list i j k sum))**

(4 2 4 19)

> (**let ([i 1] [j 1] [k 1] [sum 0])**

**(while (<= i 3)**

**(begin (set! j 1)**

**(while (<= j i)**

**(begin**

**(if (= j 2) (break 1))**

**(set! sum (+ sum i j))**

**(set! j (+ 1 j))))**

**(set! k 1)**

**(while (<= k i)**

**(begin**

**(set! sum (+ sum k))**

**(set! k (+ 1 k))))**

**(set! i (+ 1 i))))**

**(list i j k sum))**

(2 2 2 6)