CSSE 304 Exam #2 Oct 27, 2020 (Day 30.5) Computer part (the only part)

You may use the online resources for the course plus TSPL, EoPL, CSUG, your notes, and a Scheme programming environment, plus the PLC grading program and You may not use any other web or network resources. You are allowed to look at and use any Scheme code that YOU and/or YOUR TEAM have previously written.

Submit your code to the PLC grading server. There are three PLC assignments there, one for the interpreter problem, one for the CPS problem, and one for problems 1-3. I may give you a different amount of partial credit than the grading server 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 problem and spend so much time on it that you do not get to the other problems. Don't do that!

No error checking! You may assume that all test cases will be in correct format and should produce actual answers. For example, in the interpreter problem this means that you do not have to include error-checking in the new clause that you will add to `parse-exp`.

No mutation! Unless specified otherwise for a given problem, your code must not use mutation. In some cases, the test code will do mutation, but that does not mean that YOUR code should include mutation.

1. (10 points) At the beginning of the Day 20 class (feel free to look at the video and/or slides), we saw that `(map proc ls)` does not necessarily apply `proc` to the elements of `ls` in left-to-right order. The slides' example, in Chez Scheme version 9:

```
> (define a 0)
> (map (lambda (x) (set! a (add1 a)) a)
       '(a b c d e f g h i j k l m n o p))
(15 16 13 14 11 12 9 10 7 8 5 6 3 4 1 2)
```

You are to write `map-in-order`, which takes as its arguments a procedure `proc` and a single list applies proc to the list elements in left-to-right order, and returns the list of results. **You are not allowed to use any kind of sorting procedure.**

```
> (define a 0)
> (map-in-order (lambda (x) (set! a (add1 a)) a)
                '(a b c d e f g h i j k l m n o p))
(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16)
```

2. (15 points) Java and C have a `do ... while` syntax. The semantics is similar to `while`, but the test is done at the end of the loop body, guaranteeing that the loop body is evaluated at least once. Use `define-syntax` to define a semantically equivalent syntax in Scheme.

`(my-do (loop-bodies) while test)`.

It evaluates the (one or more) loop bodies in order, then evaluates the test. If the value of `test` is not `#f`, it repeats.

```
> (let ([x 5])
    (my-do ((set! x (+ 1 x))) while (not (zero? (modulo x 5))))
    x)
10
> (let ([x 4])
    (my-do ((set! x (+ 1 x))) while (not (zero? (modulo x 5))))
    x)
5
> (let ([x 4] [y 5])
    (my-do ((set! y 6)
            (set! x (+ x y))
            (my-do ((let ([x (+ 1 x)])
                        (set! x (+ x y))
                        (set! y (+ x y)))
                      (set! x (+ y x)))
                   while (< y 10)))
          while (< (+ y x) 15))
    (list x y))
(33 23)
> (let ([x 4] [y 5])
    (my-do ((set! y 6)
             (set! x (+ x y))
             (my-do ((let ([x (+ 1 x)])
                        (set! x (+ x y))
                        (set! y (+ x y)))
                       (set! x (+ y x)))
                   while (< y 20)))
           while (< (+ y x) 60))
   (list x y))
 (91 52)
```

3. (15 points) Mutation is allowed. For the `make-slist-leaf-iterator` homework problem, I provided a stack "class". You are to write a similar queue class. The `make-queue` constructor takes no arguments and creates a Scheme-procedure representation of an empty queue. If `q` is such a queue procedure, then

- `(q 'empty?)` returns a boolean value that indicates whether or not the queue is empty.
- `(q 'enqueue! obj)` adds `obj` to the rear of queue q. Does not return a value.
- `(q 'dequeue!)` removes the object that is at the front of the queue and returns its value.

The PLC server will give you up to 10 points. If you get those 10 points, we will look at your code by hand. The last 5 points are for having all three queue methods operate in constant time (no points for two out of three).

```
> (let ([q1 (make-queue)] [q2 (make-queue)])
    (q1 'enqueue! 3)
    (and (q2 'empty?) (not (q1 'empty?))))
#t
> (let ([q1 (make-queue)] [q2 (make-queue)])
    (q1 'enqueue! 3)
    (q1 'enqueue! 4)
    (q1 'dequeue!))
3
> (let ([q1 (make-queue)] [q2 (make-queue)])
    (q1 'enqueue! 3)
    (q1 'enqueue! 4)
    (q2 'enqueue! (q1 'dequeue!))
    (and (not (q1 'empty?)) (not (q1 'empty?))))
#t
> (let ([q1 (make-queue)] [q2 (make-queue)])
    (q1 'enqueue! 3)
    (q1 'enqueue! 4)
    (q2 'enqueue! (q1 'dequeue!))
    (q2 'enqueue! (q1 'dequeue!))
    (and (q1 'empty?) (not (q2 'empty?))))
#t
> (let ([q1 (make-queue)] [q2 (make-queue)])
    (q1 'enqueue! 3)
    (q1 'enqueue! 4)
    (q2 'enqueue! (q1 'dequeue!))
    (q2 'enqueue! (q1 'dequeue!))
    (let* ([x (q2 'dequeue!)]
           [y (q2 'dequeue!)])
     (list x y (q2 'empty?))))
(3 4 #t)
```

4. **(25 points).** In the `cps.ss` file, I have provided the CPS code for `memq-cps`, `intersection-cps`, and also CPS versions of several helper procedures. In this problem, we will represent continuations by Scheme procedures; the data-structures representation will be saved for the final exam. You are to rewrite the `memq-cps` code in "fully-CPSed" form. I.e., consider every procedure (except `make-cps`) to be substantial. Thus you should use `null?-cps`, `eq?-cps` , `car-cps` and `cdr-cps` and provide continuations when you apply them (in tail position, of course). In order to earn any points for this problem, your code must pass all of the `memq-cps` tests on the server. Then I will look at your code. I will subtract 8 points for each procedure call that is not in tail position.

```
> (memq-cps 'a '(b c a d) (make-k list))
(#t)
> (memq-cps 'a '( b c d) (make-k not))
#t
> (memq-cps 'a '() (make-k list))
(#f)
> (memq-cps 'a '(b c d a) (make-k not))
#f
```

(15 points Extra Credit) Also rewrite `intersection-cps` in fully-CPSed form. This extra-credit part is "all-or-nothing." In order to earn any points, you must get all of the points from the PLC server and have all procedure applications (except applications of `make-k`) in tail position. Also, your lines of code must not be super long and must be readably indented.

Warning: The ratio of possible_points to time_needed for this extra-credit problem is probably lower than for any other problem on the exam. I suggest that you do what you can on the other problems before attempting this one.

```
> (intersection-cps '(b) '(b) (make-k list))
((b))
> (intersection-cps '(b) '(a) (make-k list))
(())
> (intersection-cps '() '(b) (make-k (lambda (v) v)))
()
> (intersection-cps '(a d e g h) '(s f c h b r a) (make-k list))
((a h))
```

5. (30 points) Mutation is allowed. In the Pascal programming language, a `for` loop has two possible forms:

FOR <var> : = <expression> TO <expression> DO <statement>
FOR <var> : = <expression> DOWNTO <expression> DO <statement>

- In each form, the loop variable gets its initial value from the first expression, and then is incremented (in the first form) or decremented (in the second form) by 1 each time through the loop.
- In our semantics (unlike Pascal's semantics), the loop variable does not have to have been declared before the loop; this *for* loop syntax always creates a new local binding for the loop variable.
- The body of the loop is not allowed to mutate the loop variable. You do not have to check for this (it would be an error), but knowing that such mutation is not allowed may enable you to make your code simpler.
- Some particular Pascal compilers allowed the programmer to specify that the increment should be by a different number than 1 or -1, but this was never part of the Pascal standard, as far as I know. You do not have to implement that. The expression that determines the stopping condition for the loop (the one before the DO) is only evaluated once, before the loop begins.
- Since A16 requirements do not include implementing `set!`, my test cases do all of their mutation using the primitive procedure set-car!
- In case you did not get A16 working, it is likely that a solution can be built with working A14 code as your starting code.

For example, `FOR i := 3 TO 7 DO p(i)` calls the procedure `p` five times.
`FOR i := 6 DOWNTO 3 DO p(i)` calls the procedure `p` four times.
`FOR i := 6 TO 3 DO p(i)` never calls the procedure `p`, because 6 is already larger than 3.

If we want to add this loop construct to our interpreted language, we can (and you should) use the following syntax additions:

```
<expression> ::=  (for <var> := <expression> to <expression> do <expression>)
             ::=  (for <var> := <expression> downto <expression> do <expression>)
```

Like a `while` loop, a `for` loop is executed for effect, and so the loop does not return a value.

Example. Your interpreter is not required to execute this example, since you are not required to have `set!` working yet. But it is still an instructive example.

```
> (rep)
--> (let ([i 9] [x 4])
      (begin (for i := (+ 3 5) to (+ x 10) do (set! x (+ x 2)))
             (list x i)))
(18 9)
```

Some actual test cases for this problem.

```
> (eval-one-exp
 '(let ([list-of-num (list 0)])
    (begin
      (for i := 1 to 12 do
      (set-car! list-of-num
                (+ (car list-of-num)
                (- (* 2 i) 1))))
      list-of-num)))
(144)

>(eval-one-exp
 '(let ([list-of-num (list 0)])
    (begin
      (for i := 12 to 1 do
      (set-car! list-of-num
                (+ (car list-of-num)
                   (- (* 2 i) 1))))
      list-of-num)))
(0)
```

(continued on next page)

```
>(eval-one-exp
 '(let ([list-of-num (list 0)])
    (begin
      (for i := 12 downto 1 do
      (set-car! list-of-num
                (+ (car list-of-num)
                   (- (* 2 i) 1))))
      list-of-num)))
(144)

>(eval-one-exp
 '(let ([list-of-num (list 0)]
   [i 1000])
    (begin
      (for i := 12 downto 1 do
      (set-car! list-of-num
                (+ (car list-of-num)
                   (- (* 2 i) 1))))
      (list (car list-of-num) i))))
(144 1000)

> (eval-one-exp
 '(let ([list-of-num (list 0)])
    (begin
      (for i := (let ([list-of-num (list 0)])
        (begin
          (for i := 5 downto 1 do
          (set-car! list-of-num
                    (+ (car list-of-num)
                       (- (* 2 i) 1))))
          (car list-of-num)))
      downto 1 do
      (set-car! list-of-num
                (+ (car list-of-num)
                (- (* 2 i) 1))))
      list-of-num)))
(625)
```