CSSE 304 Exam 2 Oct 29, 2019 (day 30.5) Name_____ Section 01(9:00) 02(10:00) 03(11:00)
PART 1

Y**ou must turn in Part 1 before you pick up part 2 or use your computer.**
During the exam you may not use email, IM, or other chat tools, cell phone, PDA, headphones, ear buds, or any other communication device or software. If you have accommodations that allow one or more of these, you may use it/them.

Part 1: You will not lose points for minor syntax errors in your code, except in problem 4.
Part 1 Suggestion: Spend no more than 75 minutes on this part. No resources allowed other than a a writing implement at eraser.

Parts 2, computer. You may use TSPL and EoPL, your notes, and a Scheme programming environment plus the PLC grading program and any materials that I provided online for the course. You are allowed to use your notes and any Scheme code that **you** have previously written.

Caution! It is possible to get so caught up in getting all of the points for one problem that you do not get to the other problems. Don't do that! I will give partial credit if you have the main ideas, even if a procedure does not produce correct answers for any test cases.

Problem	Max score	Your score
1	7	
2	6	
3	3	
4	9	
5	10	
6	15	
Total	**50**	

Sign the following statement if it is true:

No one other than the instructor has given me any information about the contents of this exam. Furthermore, after I have finished this part of the exam, I will not communicate anything to anyone about the exam's contents or difficulty level until after 10 PM on October 29.

Signed: _____

1. (7 points) Consider the following lambda-calculus expression.

```
(g (lambda (a b c)
     (d (lambda (d e f)
          (e (lambda (g)
               (b f g k))))))
```

(2) How many different variables occur free in this expression? _____

(2) How many different variables occur bound in this expression? _____

(3) What is the lexical depth of each of these variables? b ___ f ___ g ____

2. (6 points) Consider the `expression` datatype from your interpreter. Executing the `define-datatype` statement defines the constructors, and `expression?` But it does not define `if-exp?`. Use `cases` to define it so that `(if-exp? obj)` returns `#t` if `obj` is an `if-exp` and `#f` otherwise. Note that `obj` can be any Scheme value. For full credit, your code must be representation-independent (i.e., don't use `car`, `cadr`, etc.). Some of the credit will be for having short and simple code. Note that this is about parsed expressions, so there is no need to check for syntax errors.

```
(define if-exp?
   (lambda (obj)
```

3. **(3 points)** In the starting interpreter code that I gave you, the `proc-val` datatype was defined in the code before the `environment` datatype. When you added `lambda` to the interpreted language, it was necessary to switch the order of those two `define-datatype`s because the closure variant of `proc-val` has an `environment` field..
However, it the second datatype had been called `environ` instead of `environment`, the change of code ordering would not have been necessary. Explain this phenomenon briefly.

4. **(9 points)** In our interpreters, application of most of the primitive procedures in `apply-prim-proc` can and should be implemented by simply applying the corresponding Scheme procedure to the arguments. But there are a few prim-procs that either cannot or should not (because there is a simpler way) be implemented that way. For each of the following primitive procedures, fill in the code for a correct and efficient implementation (an implementation whose code is as short as possible). Do not call the corresponding Scheme procedures. **For this problem only, exactly correct code is required for full credit.**

```
(define apply-prim-proc
   (lambda (prim-proc args)  ; args is a list of Scheme values.  prim-proc is the symbol that names the primitive procedure.
      (case prim-proc        ; many cases are omitted; I only show the ones  that you should implement here.
        [(list)




        [(apply)




        [(procedure?)
```

5. **(10 points)** Use `define-syntax` to define a simple `for` loop, as illustrated by the examples on the next page. Each time through the loop, the loop variable is incremented by 1; when the value of the variable is greater than the end value, the loop stops. What the loop returns is unspecified (the user should only write code that does not depend on what it returns, do it doesn't matter what, if anything, your code returns). The loop variable is bound to the initial value by the `for` loop; changing its value does not change the value of any enclosing variable with the same name.
The examples all involve mutation (what else would you do with a loop?) , but the implementation of the for-loop itself does not have to use mutation (you are allowed to use mutation if you wish. Write your answer below.
Hint: There is a short and uncomplicated solution.

```
(define-syntax for
```

Did you sign the statement on page 1?

for-loop examples. Note: there are circumstances that call for a more complex solution, but if your code would work for all of these examples, it should be okay.

```
> (let ([sum (list 0)])
    (for i from 2 to 5
       (set-car! sum (+ i (car sum))))
    (car sum))
14
> (let ([sum (list 0)]
        [i 8])
    (for i from 2 to 5
       (set-car! sum (+ i (car sum))))
    (list i (car sum)))
(8 14)
> (let ([sum (list 0)])
    (for i from 2 to 7
       (if (even? i)
         (set-car! sum (+ i (car sum)))))
    (car sum))
12
> (let ([sum (list 0)])
    (for i from 2 to 1
       (set-car! sum (+ i (car sum))))
    (car sum))
0
> (let ([f (lambda (n)
           (let ([sum (list 0)])
             (for i from n to (* 2 n)
                (set-car! sum (+ i (car sum))))
           (car sum)))])
  (f 3))
18
```

Did you sign the statement on page 1?

6. (15 points) Show the code needed to add the new construct from the previous problem to the language interpreted by your interpreter. You can earn most or all of the points without having precisely correct syntax, but it should be close enough so that I can see that you know all of details that need to be done. A bit less partial credit will be given for code that is nearly correct but leaves out important details. Little or no partial credit will be given for "Here's what needs to be done, but I don't know how to write the code." This problem is about knowing and understanding your interpreter code very well.

Your code does not have to check for syntax errors. A solution should be doable starting with either an A14 interpreter or an A16 interpreter. Possibly even with an A13 interpreter; you can choose the one that works best for you as a starting point. It can be done with no mutation in your code, but that is not a requirement for this problem. Make sure that you re-read the description and examples from the previous problem.

On the next page, I have a grid where you should write your answers. In each part, you only need to show the code that you would add; you only need to show context if it is necessary to understand what you wrote. If you do not change one of the procedures that I listed, simply write "No changes" in that box. If necessary, we can look at you submitted code on the PLC server to see the context. Also, if you did not change one of the procedures I listed but did add or change another procedure with lots of code, feel free to cross out and replace the procedure name that I put in the first column, so you have a place that has more room to write in the second column.

If there is anything about the way that you and your partner organized your interpreter that may help us to understand the code you write on the next page, feel free to describe it below.

Problem 6 answer: Your starting interpreter: A13 ____ A14 ____ A16 ____

Show the additions to each of the following (or write No Changes). No error-checking is required. You may need to write small.

expression datatype	
parse-exp	
syntax-expand	
eval-exp	
other procedure	
other procedure	
other procedure	