**CSSE 304 Exam 2 Feb 5, 2020 (day 30.5) Name\_\_\_\_\_\_SOLUTION\_\_\_\_\_\_\_\_ Section 01(10:00) 02(11:00) 03(12:00)**

**PART 1**

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 Suggestion:** Spend no more than 75 minutes on this part. No resources allowed other than a writing implement and eraser.

**Part 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.

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, I will not communicate anything to anyone about the exam’s contents or difficulty level until after 10 PM on February 5.

Signed: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |  |
| --- | --- | --- |
| **Problem** | **Max score** | **Your score** |
| 1 | 10 |  |
| 2 | 10 |  |
| 3 | 3 |  |
| 4 | 8 |  |
| 5 | 10 |  |
| 6 | 5 |  |
| 7 | 4 |  |
| **Total** | **50** |  |

1. **(10 points)** Consider the following lambda-calculus expression. Fill in the table for the occurrences of each variable:

(lambda (a c e)

(lambda (b c d)

(lambda (a d)

(a b c d e)))))

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | a | b | c | d | e |
| Lexical depth | **0** | **1** | **1** | **0** | **2** |
| Lexical position | **0** | **0** | **1** | **1** | **2** |

One point for each correct answer

1. **(10 points)** Based on the following code, write comb-cps in proper CPS form, using “scheme procedure” continuations. All calls to fact-cps must be in tail position. You may assume that fact-cps and apply-k have already been written, and that – and / are primitives (can be called in non-tail position). To save time, you are not required to call make-k.

(define comb (n m)

(/ (fact n) (fact m) (fact (- n m))))

(define comb-cps ; you fill in the rest

The calls to fact-cps can be done in a different order. Calling make-k and apply-k is not required, since continuations are Scheme procedures.

Check carefully to make sure that all calls to fact-cps and the application of the continuation are in tail position.

**(lambda (n m k)**

**(fact-cps n**

**(make-k (lambda (n-v)**

**(fact-cps m**

**(make-k (lambda (m-v)**

**(fact-cps (- n m)**

**(make-k (lambda (n-m-v)**

**(apply-k k**

**(/ n-v m-v n-m-v))))))))))))**

1. **(3 points)** Is the following statement true? (circle one) Yes **No**. Explain your answer below.

**Statement:** When Scheme evaluates a lambda-expression, it creates a closure, then immediately uses that closure to   
 build an extended environment that will be used when the body is executed.  
**Evaluating the lambda-expression just makes and returns the closure.** Full credit for NO answer and reasonable explanation. 2 points for NO answer without reasonable explanation. 0 points for YES answer.

1. **(8 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.**You are not allowed to use Scheme’s list procedure in your implementation of list. No such restriction on the others.

(define apply-prim-proc

(lambda (prim-proc args) **;** args **is a list of Scheme values.**

(case prim-proc ; prim-proc **is the name of the primitive procedure.**

[(list) **args ; many cases are omitted;**

**; I only show ones that you must implement** .

[(apply) **(apply-proc (car args) (cadr args))]**

**One point for cdr instead of cadr**

[(procedure?) **(proc-val? (car args))]**   
 ; no credit unless student used proc-val?

[(map) **(map (lambda (x)**

**(apply-proc (car args) (list x))**

**(cadr args))]**  
 This is for mapping a procedure that takes only one argument.   
 Students may do the more general case or they may do the single case in a different way.   
 Read their code carefully to check for correctness.

1. **(10 points)** I am not sure if my definition of flipflop exactly matches any common electronic device. But it should be easy to see how it works.   
   (make-flipflop name bool-value) creates a flipflop with the given name and value.   
   (toggle-flipflop name) toggles the value of the flipflop with that name.   
   (flipflop-value name) returns the current value of that flipflop. Here is a transcript:

> (load "flipflop.ss")

> (make-flipflop ff1 #t)

> (make-flipflop ff2 #t)

> (make-flipflop ff3 #f)

> (make-flipflop ff4 #t)

> (toggle-flipflop ff2)

> (toggle-flipflop ff4)

> (map flipflop-value (list ff1 ff2 ff3 ff4))

(#t #f #f #f)

> (toggle-flipflop ff3)

> (map flipflop-value (list ff1 ff2 ff3 ff4))

(#t #f #t #f)

Show code that could be the contents of the file *flipflop.ss*, code that would make the above transcript be valid. Your code must not contain any of the symbols ff1, ff2, ff3, or ff4.

The main point of this problem is that make-flipflop and toggle-flipflop cannot be procedures. If the student writes either as a procedure and doesn’t say “this won’t work” or something like that, 0 points.

If student recognizes that procedures won’t work but does not think of using define-syntax, 5 points.

flipflop-value must be a procedure because it is used with map. -2 if student tries to use define-syntax with that.

Take off one or two points for any other details that are not correct.

**(define-syntax make-flipflop**

**(syntax-rules ()**

**[(\_ var val)**

**(define var val)]))**

**(define-syntax toggle-flipflop**

**(syntax-rules ()**

**[(\_ var)**

**(set! var (not var))]))**

**(define flipflop-value**

**(lambda (var) var))**

1. **(5 points)** In class, we discussed 3 approaches to creating the effect of the “circular environment” needed to implement letrec.  
   1. (2) Circle the approach that you used: no mutation mutation syntax-expand other

Look at the student’s submitted code on-line to determine the correct answer. For the student who did not submit A16, look at A17 code.

* 1. (3) Describe a different approach in enough detail (English and/or code) to convince me that you know how it works. If you used the no-mutation approach, describe the mutation approach. Otherwise describe the no-mutation approach.

**Mmutation:** A normal extended-env record is used. So the real work is done in extend-env-recursively. When we build the environment, we put in the list of variable names and we make a vector that will contain the closures. Now we have the letrec environment. We build each of the closures, making its environment pointer point to this new environment, and we use vector-set! to put each closure into its correct place in the vector.

Students can use code in their explanation, but there should be some words that indicate that they understand that code.

(define extend-env-recursively   
 (lambda (proc-names idss bodiess old-env)   
 (let ([len (length proc-names)])   
 (let ([vec (make-vector len)])   
 (let ([env (extended-env-record   
 proc-names   
 vec   
 old-env)])   
 (for-each (lambda (pos ids bodies)   
 (vector-set! vec   
 pos   
 (closure ids   
 bodies   
 env)))   
 (iota len)   
 idss   
 bodiess) ; end of for-each  
 env)))))

**No mutation:** The recursively-extended-env-record does not contain the actual closures that would be obtained by evaluating the lambda expressions. Instead it contains the *ingredients* for building each of the closures. The real work is done in apply-env. When we look up a variable in the letrec environment, it then builds the correct closure and returns it.

Students can use code in their explanation, but there should be some words that indicate that they understand that code.

Students do not need to include code. But I include the code here to help the grader understand what is happening and how to interpret what the student writes.

(define apply-env

(lambda (env sym)

(cases environment env

; other cases go here

[recursively-extended-env-record   
 (procnames idss bodiess old-env)   
 (let ([pos   
 (list-find-position sym procnames)])   
 (if (number? pos)

(closure (list-ref idss pos)   
 (list-ref bodiess pos)

env)   
 (apply-env old-env sym)))])))

env is the letrec environment itself, and we want the env pointer in the closures to point to that.

1. **(4 points)** My solution for the in-class E&C exam is on the next page. I drew the diagram assuming that each application of c *will* create a new environment; you should assume the same thing when you answer this question. In that solution, there are four closures and six environments.

If the code in the define form is the same as the previous example,   
but the second expression is changed to

(((curry (lambda(z t) (\* z t 2))

5)

7)

How many closures \_\_**5**\_\_ and environments \_\_**8**\_\_   
will be created during the execution of the entire code?

One extra (lambda (x) … ) closure produced by the additional call to c.

One extra env that binds n and args, from the additional application of c.

One extra env that binds x, from the application of the additional closure.

Two points for each part. Full credit for correct numbers. Perhaps 1 point for an explanation that seems somewhat on the right track.

