CSSE 304 Programming Language Concepts
Final Exam
Winter 2024/25

Name: _____

The final is closed book, closed notes, closed code, except, for the programming portion, you may look at the Racket documentation. You may not search the web and you may not use genAI tools of any kind. You must turn in the written portion before beginning work on the programming portion.

By signing below, I certify that (i) all the work for this exam is my own work, (ii) I did not use any materials for the written portion and (iii) I did not use search engines or GenAI tools to look-up or produce answers for this exam. If you do not sign below, you will be assigned a zero (0) for the entire exam.

Your signature: _____

Part I: Written portion

1) [5 points] a) What value is returned by the following code?

```
(+ 3 (let ([a (+ 4 (call/cc (lambda (k) (+ 4 (k 5) 6))))]) (+ 7 a)) 8)
```

b) What continuation is assigned to `k`? Either describe it or name the one of the cps-interpreter we have seen in class.

2) [7 points] Consider the following code.

```
(define state #f)
(define fac (lambda (n)
              (if (= n 0)
                  (begin (call/cc (lambda (k) (set! state k))) 1)
                  (* n (fac (- n 1))))))
```

a) Assume the user runs: `(fac 5)` first. What is the value returned when one then runs the following code? `(state 2)`

b) What continuation is assigned to `k`? Either describe it or name the one of the cps-interpreter we have seen in class.

3) [10 points] Consider the following code.

```
(let ([x1 (call/cc (lambda (k) k))]
      [x2 (call/cc (lambda (k) k))])
  (display 1)
  (x2 (lambda (x) "hi"))
  (display 2)
  (x1 (lambda (x) "hi")))
```

a) What happens when the interpreter executes the code surrounded by the **solid** boxed line?

b) What happens when the interpreter executes the code surrounded by the **dashed** boxed line?

c) What is displayed by this code?

4) [8 points] Consider the following code.

```
((lambda (x) x) (call/cc (lambda (k) k)))
```

b) What is the value returned when the code is run?

b) What continuation is assigned to `k`? Either describe it or name the one of the cps-interpreter we have seen in class.

Part II. Programming portion.

5) [20 points] Convert the following procedure into tail form, using data-typed continuations. Please call the tail-form version: **`occurs-free?-cps`**

```
(define occurs-free?
  (lambda (var exp)
    (cond [(symbol? exp) (eqv? var exp)]
          [(eqv? (car exp) 'lambda)
           (and (not (eqv? (caadr exp) var))
                (occurs-free? var (caddr exp)))]
          [else (or (occurs-free? var (car exp))
                    (occurs-free? var (cadr exp)))])))
```

6) [10 points] Syntax expand the following form into nested if-expressions. Add the following form to your *recursive interpreter*.
(**`unpack <list> <two-arg-predicate> <literal>)`** takes a list of literals, a two-argument procedure and a literal. Unpack checks whether the property (as specified by the procedure and element) holds of any elements of the list. It returns the first element of the list for which the property holds or #f.

```
(unpack '(1 2 3 4 5) eq? 5)                              -> 5
(unpack '(1 2 3 4 5) > 0)                                -> 1
(unpack '() eq? 5)                                       -> #f
(unpack '(1 2 3 4 5) (lambda (x y) (> (* x y) 2)) 2)     -> 2
```

7) [15 points] Convert your *recursive interpreter* so that the two armed `If` is a primitive. Please disable the single armed `if`. We should then be able to do things as follows. Partial credit depends on how far you get into this problem. In a comment near the top of your submission file, please explain the major drawback of making IF a primitive.

```
(let ([if 42]) if)           -> 42
(let ([if 42]) (if 3 4 5))   -> error
(if 3 4 5)                   -> 4
(if #f 4 5)                  -> 5
```

8) [25 points] Add **delay** and **force** to your *recursive interpreter*. Please ensure you add it to eval-expression (or your equivalent.) The syntax for **force** and **delay** are given below. **`Delay`** causes that the expression passed to it is not evaluated when processed by eval-expression. Typically, the entire delay expression is assigned to some variable. **`Force`** forces the evaluation of a delayed expression. Notice that force takes a symbol, you may assume that the symbol is a variable to which a delay-expression has been assigned. Please note that the expression being delayed is evaluated only once.

```
<exp> ::= (delay <exp>)
<exp> ::= (force <symbol>)
```

Examples:

```
(let ([x (delay 2)])
   (force x))                    -> 2

(let ([x 3])
   (let ([p (delay
              (begin (set! x (+ x 1)) x))])
     (begin
       (set! x (+ 3 x))
       (let ([q (list (force p))])
          (cons (force p) (cons (force p) q))))))    -> (7 7 7)
```