CSSE 304 Exam #1 Part 1 Jan 8, 2020 (day 14.5) Name___SOLUTION_____ Section 01 (10:00) 02 (11:00) 03 (12:00)

Problem	Possible	Earned
1	10	
2	5	
3	5	
4	4	
5	6	
Total	30	

You must turn in Part 1 before you use your computer for anything. During the entire exam you may not use email, IM, phone, tablet, headphones, ear buds, or any other communication device or software. Except where specified, efficiency and elegance will not affect your scores, provided that I can understand your code.

On both parts, assume that all input arguments will be of the correct types for any procedure you are asked to write; you do not need to check for illegal input data.
Mutation is not allowed in code that you write for this exam.

Part 1, written. Allowed resources: Writing implement.
Suggestion: Spend no more than 40 minutes on this part, so that you have a lot of time for the computer part. 30 minutes is ideal.

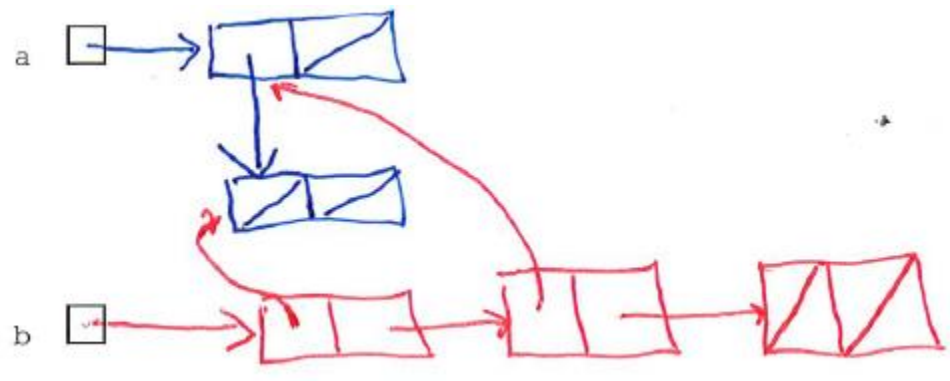
Built-in procedures & syntax that are sufficient for this paper part of this exam:

Procedures:
Arithmetic: +, - , *, /, modulo, max, min, =, <, ≤, >, ≥
Predicates and logic: not, eq?, equal?, null?, zero?, procedure? positive?, negative?, pair?, list?, even?, odd?, number?, symbol?, integer?, member
Lists: cons, list, append, length, reverse, set-car!, set-cdr!, car, cdr, cadr, cddr, etc.
Functional: map, apply, andmap, ormap
Homework: Any procedure that was assigned for A01-A08.

Syntax:
lambda, including (lambda x …) and
(lambda (x y . z) …),
define, if, cond, and, or, let, let*, letrec, named let, begin, set! (You may not use mutation in your code unless a specific problem says you can).

1. **(10 points, 2 for each part)** Consider the execution of the code below. Draw the box-and-pointer diagrams that represent the results of the two `define`s. Then show what Scheme would output from the execution of each of the last three expressions.
   Be careful! "Almost correct" answers will usually receive no partial credit. Note that the last three parts can be done even if you cannot draw the two diagrams correctly.

```
> (define a '(((()))))
> (define b (list (car a) a (cdr a)))
```




```
> b
```

((()) ((())) ())

```
> (procedure? ((lambda (x) (lambda (y) (+ x y))) 5))
```

#t

```
> (procedure? and)
```

syntax error *or* exception **(one point for answering #f)**

2. (5 points) In assignment A6, you were asked to write `curry2`. The `curry2` procedure takes as its argument a procedure `f` that expects two parameters. Then `(curry2 f)` returns a procedure that takes the two parameters one at a time. My solution:

```
(define curry2
  (lambda (f)
    (lambda (x)
      (lambda (y)
        (f x y)))))
```

For this problem, you must write an inverse for `curry2`. `uncurry2` takes a curried procedure and produces a corresponding procedure of two arguments. If `g` is any procedure for which `((g a) b)` makes sense, then the return value of `((uncurry2 g) a b)` is equal to the return value of `((g a) b)`. Another way of saying the same thing is that if **`f`** is a procedure for which `(f a b)` makes sense, then `((uncurry2 (curry2 f)) a b)` is equal to `(f a b)`.

Examples:

```
(let ([f (lambda (x)
           (lambda (y)
             (list y x)))])
  ((uncurry2 f) 6 7))                          ➔ (7 6)

((uncurry2 (curry2 cons)) 4 '())               ➔ (4)
```

You are to fill in the rest of the definition of `uncurry2`. Be *very careful* about parentheses placement.
Incorrect parens will probably result in zero points for this problem.

```
(define uncurry2      ; The correct answer is almost entirely about where the parentheses go.
  (lambda (f)
    (lambda (a b)
      ((f a) b))))
```

Of course the parameters don't have to be called f, a and b.
If there are 3 or 5 parentheses at the end, 4 points.
Any other error: 0 points.

3. (5 points) Given the definitions in the box at the right:

```
(define atoi ; not related to the C function that has the same name
  (lambda (n)
    (if (zero? n)
        '()
        (cons (- n 1) (atoi (- n 1))))))
(define mystery
  (lambda (m  n)
    (map (lambda (x) (+ m x))
         (atoi n))))
```

What is the value of `(mystery 4 3)`? **(6 5 4)**

`(atoi 3)` produces `(2 1 0)`.
Then the call to map produces a list with 4 added to each of those numbers.

3 points for any of the following answers:

(4 4 4) (6 6 6) (6 5 4 3) (5 4 3) (5 4 3 2)

1 or 2 points for any other answer that seems to display a degree of understanding.

4. (4 points) Here is the stack "object" code from Assignment 9:
We discussed the following questions in class.

```
(define make-stack
 (lambda ()
  (let ([stk '()])
   (lambda (msg  . args )
    (case msg
      [(empty?) (null? stk)]
      [(push)   (set! stk (cons (car args) stk))]
      [(pop)    (let ([top (car stk)])
                  (set! stk (cdr stk))
                  top)]
      [else (errorf 'stack
                    "illegal message")])))))
```

How will stacks behave differently if we
(a) move the `let` so it comes **before** both `lambdas` (and leave the rest of the code unchanged)?
All stacks share the same `stk` object,
so in effect there is only one stack
A call to make-stack make a new reference to that stack.

(b) move the `let` so it comes **after** both `lambdas` (and leave the rest of the code unchanged)?

Every call to a stack "method" first creates a new empty stack, so the value of the `stk` field is not persistent.

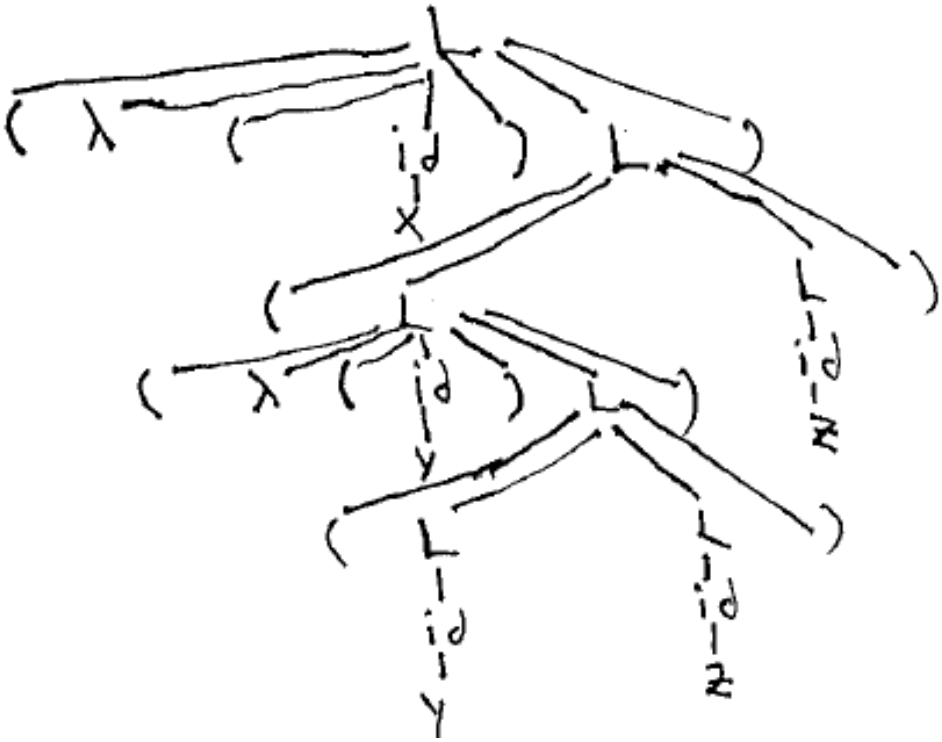
```
<LcExpr> ::=
 <identifier> |
 (lambda (<identifier>) <LcExpr>) |
 ( <LcExpr> <LcExpr> )
```

5. (6 points) Our original grammar for lambda-calculus expressions:

Consider the expression `(lambda (x) ((lambda (y) (y z)) z))`

(a) **(2 points)** In that expression, which variables occur bound? ____**y**_____ occur free? _____**z**_____
No partial credit for this part

(b) **(4 points)** Draw the derivation tree for that expression. To make it easier to draw this (as I did on the whiteboard in the Day 11 class), you are allowed to write **L** in place of `<LcExpr>`, **λ** in place of `lambda`, and **id** in place of `<identifier>`.



2 points if the student seems to have the idea of how derivations work, and if they use the three specific rules from this grammar. The other 2 points are for the correct tree. Be generous with partial credit. An answer with one or two errors should receive 3 or 4 points.

Computer part:

```
(define (notate-depth slist)  ; This is in live-in-class floder, day 8;
  (let notate ([slist slist]
               [depth 1])
    (cond [(null? slist) '()]
      [(symbol? (car slist))
       (cons (list (car slist) depth)
         (notate (cdr slist) depth))]
      [else ; car is an s-list
       (cons (notate (car slist) (+ 1 depth))
         (notate (cdr slist) depth))])))

(define  (notate-depth-and-flatten slist)
  (let ([slist (notate-depth slist)])
    (let flatten ([slist slist]) ; very similar to the flatten procedure from day 8.
      (cond [(null? slist) '()]
        [(null? (car slist)) (flatten (cdr slist))]
        [(symbol? (caar slist))
         (cons (car slist)
           (flatten (cdr slist)))]
        [else ; car is an s-list
         (append (flatten (car slist))
             (flatten (cdr slist)))]))))

(notate-depth-and-flatten '())
(notate-depth-and-flatten '(a ((b))))
(notate-depth-and-flatten '((a () (b c (() d))) e))

; for full credit, must be O(N)
(define prefix-sums
  (lambda (lon)
    (let prefixes ([lon lon] [sum 0])
      (if (null? lon)
      '()
      (cons (+ (car lon) sum)
        (prefixes (cdr lon)
              (+ (car lon) sum)))))))


(prefix-sums '(1 2 3 4 5 6))
(prefix-sums '(1 3 5 2))
(prefix-sums '(4))
(prefix-sums '(3 2 0 -5 6))


(define suffix-sums
  (lambda (lon)
    (car
     (let suffixes ([lon lon])
       (if (null? lon)
       (list '() 0)
       (let ([result (suffixes (cdr lon))])
         (list (cons (+ (car lon) (cadr result))
             (car result))
           (+ (car lon) (cadr result)))))))))

(suffix-sums '(1 2 3 4 5 6))
(suffix-sums '(1 3 5 2))
(suffix-sums '(4))
(suffix-sums '(3 2 0 -5 6))
```

```
; for full credit, must be O(N).
(define evens-odds
  (lambda (los)
    (let splitter ([los los] [evens '()] [odds '()])
      (cond
       [(null? los) (list (reverse evens) (reverse odds))]
       [(null? (cdr los)) (list (reverse (cons (car los) evens))
                                (reverse odds))]
       [else (splitter (cddr los)
                       (cons (car los) evens)
                       (cons (cadr los) odds))]))))

(evens-odds '())
(evens-odds '(b))
(evens-odds '(c d))
(evens-odds '(a b c d e f g))
(evens-odds '(b c d e f g))

(define free-occurrence-count
  (lambda (exp)
    (length (free-occurrence-list exp))))


(define free-occurrence-list ; identical to free-vars from A10,
  (lambda (exp)              ; except replace "union" with "append"
    (cond [(symbol? exp) (list exp)]
      [(eq? (car exp) 'lambda)
       (remove (caadr exp) (free-occurrence-list (caddr exp)))]
      [else (append (free-occurrence-list (car exp))
            (free-occurrence-list (cadr exp)))])))

(free-occurrence-count '(x (x y)))
(free-occurrence-count
 '(lambda (x) (lambda (x) (x y))))
(free-occurrence-count
 '(lambda (x) (lambda (y) (x (x (y y))))))
(free-occurrence-count
 '(lambda (x) (lambda (w) (x ((x z) (y ((x y) (y w))))))))
(free-occurrence-count
 '((lambda (x) (y (lambda (y) (x y))))
   (lambda (y) ((x x) (lambda (x) ((z z) (y (x x))))))))

; n will be at least 1
(define curry  ; This is a lot like the general version of compose
  (lambda (n f)
    (letrec ([curry-help
              (lambda (n args)
               (if (zero? n)
                 (apply f (reverse args))
                 (lambda (x)
                  (curry-help (- n 1) (cons x args)))))])
      (curry-help n '()))))

(let ([+-c (curry 3 +)])
  (((+-c 2) 4) 7))
(let ([cons-c (curry 2 cons)])
  ((cons-c 4) '(5 6)))
(let ([car-c (curry 1 car)])
  (car-c '(1 2 3)))
(let ([+-c (curry 7 +)])
  (((((((+-c 2) 4) 6) 8) 10) 12) 14))
(let ([+-c (curry 12 +)])
  ((((((((((((+-c 1) 2) 3) 4) 5) 6) 7) 8) 9) 10) 11) 12))
```