CSSE 304 Exam 2 Jan 22, 2019 (day 26.5) Name_____SOLUTION____ Section 01(9:00) 02(9:55) 03(10:50)

1. (10 points) Consider the following lambda-calculus expression. Fill in the table for the occurrence 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**

2. (3 points) Consider the datatype definition in the box at the right. What are the names of the procedures that are defined when that code is executed?

bintree? leaf-node interior-node (1 point each)

```
(define-datatype bintree bintree?
  [leaf-node
    (datum number?)]
  [interior-node
    (key symbol?)
    (left bintree?)
    (right bintree?)])
```

(4 points) `interior-node?` is not one of those procedures. Using `cases` in your code, define this function so that `(interior-node? obj)`, where `obj` can be any valid Scheme data object, returns `#t` if `obj` is a valid interior-node and `#f` otherwise. Your code must be representation-independent (i.e, you cannot use `car` or `cdr` in your code). Some of the credit for this part will be for having short and simple code.

```
(define interior-node?
      (lambda (obj)
        (and (bintree? obj)
         (cases bintree obj
            [interior-node (key left right) #t]
            [else #f]))))
```

Instead of `else`, there could be a specific case for `leaf-node`.

1 point for checking to make sure it is a bintree, 3 points for the rest.
On the day we introduced `define-datatype`, and for the next couple of class days, I discussed *Chez* Scheme's transparent implementation of the datatype ADT. I said (paraphrasing here) , "The representation of datatypes as lists is very helpful for debugging, but you should never use it in your code; your code must be representation-independent. Whenever you use a datatype object, you must never use `car`, `cadr`, `list?`, `list`, etc, but always use `cases`. In the statement of this problem, I emphasized this.

3. (12 points) A stack ADT could be defined to have 5 operations (you should implement each of them here). Note that this is a different interface than the "OO" stack that we saw earlier.

```
new-stack    ; creates a new stack
empty?       ; tests for empty stack
push!        ; pushes an object onto stack.
pop!         ; removes top element and returns it.
top          ; returns top element without removing it.
```

2 points for each procedure

3 points for each syntax

A Scheme list seems like a simple enough way to represent a stack. The car of the list is the top of the stack. Show the implementations of each of the five operations. If you can't do this, do your best to explain why.

```
> (define s1 (new-stack))
> (define s2 (new-stack))
> (empty? s1)
#t
> (push! 'a s1)
> (push! 'b s1)
> (push! 'c s1)
> (push! 'd s2)
> (top s1)
c
> (pop! s1)
c
> (push! (pop! s1) s2)
> (top s2)
b
> (top s1)
A
```

new-stack

```
(define new-stack
  (lambda () '()))
```

top

```
(define top car)
```

empty?

```
(define empty? null?)
```

`push!` and `pop!` have to be syntax, not procedures. And they have to use mutation. A student can earn one point for each by simply saying something like "This won't work because arguments are passed to procedures by value." Zero points if student simply writes these as procedures with no disclaimer that says that can't work. Even if define-syntax is used for `push!` and `pop!`, at most one point for each if mutation is not used. For `pop!`, at most two points if student does not use let or some other mechanism for saving the old value of top so if can be returned.

push!

```
(define-syntax push!
  (syntax-rules ()
    [(_ value-to-push name-of-stack)
        ;name-of-stack must be a symbol
     (set! name-of-stack (cons value-
to-push name-of-stack))]))
```

pop!

```
(define-syntax pop!
  (syntax-rules ()
    [(_ name-of-stack)
     (let ([temp (car name-of-stack)])
       (set! name-of-stack
             (cdr name-of-stack))
       temp)]))
```

```
(define counter-maker
  (lambda (f)
    (let ([count 0])
      (lambda args
      (if (and (= (length args) 1)
               (eqv? (car args) 'count))
          count
          (begin
            (set! count (+ 1 count))
            (apply f args)))))))
```