EXAM PREPARATION SECTION 8

SCHEME AND TAIL RECURSION

April 11 to April 12, 2018

1 Scheme

1. **What Would Scheme Do?** Write what a Scheme interpreter would print after each of the following expressions are entered.

```
> (let ((x 2) (y 5)) (and #f (/ 1 (- x 2) )))

> (null? (define x '(1 2 3)))

> (length '(cons 1 (cons 2 (cons 3 (cons 4 nil)))))

> (length ''(cons 1 (cons 2 (cons 3 (cons 4 nil)))))

> (append '(1 2 3) '(4 5 6))

> (append '(1 2 3) 4)

> (cond
>       ((pair? x) (cons 5 x))
>       ((list? x) (cdr x))
> )
```

How many parentheses, at minimum, should an argument to `caddaadr` contain?

```
(define (caddaadr x)
        (car (cdr (cdr (car (car (cdr x))))))
)
```

2. **deeval** Implement `deeval`, which takes in an integer `num` and another integer `k` and returns the number of ways to make an expression of the form `(_ k (_ k-1 ... (_ 1 0))))`, where each _ is either + or *, that evaluates to num. Hint: Scheme has a ”modulo” operator

```
; (deeval 1 1)
; 1
; Note: The expressions are as follows.
;               - (+ 1 0) yes
;               - (* 1 0) no
;
; (deeval 3 2)
; 1
; Note: The expressions are as follows.
;               - (+ 2 (+ 1 0)) yes
;               - (* 2 (+ 1 0)) no
;               - (+ 2 (* 1 0)) no
;               - (* 2 (* 1 0)) no
;
; (deeval 5 3)
; 2
; Note: The expressions are as follows.
;               - (+ 3 (+ 2 (+ 1 0))) no
;               - (+ 3 (* 2 (+ 1 0))) yes
;               - (+ 3 (+ 2 (* 1 0))) yes
;               - (+ 3 (* 2 (* 1 0))) no
;               - (* 3 (+ 2 (+ 1 0))) no
;               - (* 3 (* 2 (+ 1 0))) no
;               - (* 3 (+ 2 (* 1 0))) no
;               - (* 3 (* 2 (* 1 0))) no
(define (deeval num k)
    (cond
        (____________________________________________________________ 1)

        (____________________________________________________________ 0)
        (else
            (+
                (if ______________________________________________________

                    ______________________________________________________
                    0
                )
                __________________________________________________________
                )
            )
        )
)
```

3. **num-calls** Implement `num-calls`, which takes in an expression `expr` and returns a pair of integers. The first integer is the number of calls that are made to `scheme_eval` while evaluating the expression. The second integer is the number of calls that are made to `scheme_apply`. Hint: The built-in procedure **`eval`** returns the value of an expression. Only these special forms (and no user-defined functions) need be supported:
- **`if`** with both an if and an else case
- **`and`**

```
; (num-calls 1) -> expect (1 . 0)
; (num-calls '(+ 2 2)) -> expect (4 . 1)
; (num-calls '(if #f 3 4)) -> expect (2 . 0)

; Take two pairs of integers and add them elementwise.
(define (pair-add p1 p2) _____________________________________________________ )
; Return the length of a list.
(define (len lst) ____________________________________________________________ )
(define (cadr lst) (car (cdr lst)))
(define (caddr lst) (car (cdr (cdr lst))))
(define (cadddr lst) (car (cdr (cdr (cdr lst)))))

(define (num-calls expr)
    (cond
        ((not (pair? expr)) _____________________________________________________)
        ((eq? (car expr) 'if)
            (pair-add

                ___________________________________________________________________


                (if ______________________________________________________________

                        __________________________________________________________

                        __________________________________________________________

                   )
              )
        )
        ((eq? (car expr) 'and)
            (if (null? (cdr expr))

                __________________________________________________________________
                (pair-add
                      ____________________________________________________________


                      (if ________________________________________________________

                          ________________________________________________________

                          ________________________________________________________
                    )
                )
            )
        )
        (else ______________________________________________________________ )))
```

4. **Tail Recursion**

Which of the following functions are tail-recursive?

```
(define (f1)
        (or (f1) (f1))
)

(define (f2)
        (cond
                ((= x 1) (f2))
                (else 5)
        )
)

(define (f3)
        (let (x 5) (f3))
)

(define (f3)
        (if (= x 0) (f3) (cons 1 2))
)
```

Implement `isset` so that it's tail-recursive. `isset` should return `true` if the list of numbers represents a valid set or the last repeated number if not. The numbers are all positive and appear in increasing order.

```
(define (isset lst)
    (define (helper ______________________________________)
        (if (null? lst)
            ________________________________________________
            (helper
                ____________________________________________

                ____________________________________________

                ____________________________________________
                )
          )
    )
    (helper ________________________________________________)
)
```

5. **Where's Groot? (Fall 2014 Mock Final 4b)** Implement `deep-reverse`, which takes in a Scheme list and reverses the entire list, all sublists, all sublists within that, etc. **Hint**: You can use the **`list?`** operator to determine whether something is a list.

```
STk> (deep-reverse '(foo bar baz))
(baz bar foo)
STk> (deep-reverse '(1 (2 3) (4 (5 6) 7)))
((7 (6 5) 4) (3 2) 1)

(define (deep-reverse lst)

    (cond ____________________________________________________

        ______________________________________________________

        ______________________________________________________

        ______________________________________________________

        ______________________________________________________

        ______________________________________________________

        ______________________________________________________
        )
)
```