Exam Prep Section 8 Sols. - CS61A Spring 2018

Worksheet 8: Scheme & Tail Recursion

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))))

#f

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

#f

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

3

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

2

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

(1 2 3 4 5 6)

(append '(1 2 3) 4)

(1 2 3 . 4)

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

(5 1 2 3)

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

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

6 parentheses

2. **deeval**

```
; Fill in the following function definition. deeval should accept an integer num
; and another integer k and return 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
        ((= num 0) 1)
        ((or (< num 0) (= k 0)) 0)
        (else
            (+
                (if (= (modulo num k) 0)
                    (deeval (/ num k) (- k 1))
                    0
                )
                (deeval (- num k) (- k 1))
            )
        )
    )
)
```

3. **num-calls**

Fill in the following function definition. num-calls should accept an expression as input and return 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. Only these special forms (and no user-defined functions) need be supported:

- "if" with both an if and an else case
- "and"

Hint: The built-in procedure "eval" returns the value of an expression.

```
; Take two pairs of integers and add them elementwise.
(define (pair-add p1 p2)
        (cons (+ (car p1) (car p2)) (+ (cdr p1) (cdr p2)))
)

; Return the length of a list.
(define (len lst)
  (if (null? lst) 0 (+ 1 (len (cdr 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)) '(1 . 0))
                ((eq? (car expr) 'if)
                        (pair-add
                                (num-calls (cadr expr))
                                (if (eval (cadr expr))
                                        (num-calls (caddr expr))
                                        (num-calls (cadddr expr))
                                )
                        )
                )
                ((eq? (car expr) 'and)
                        (if (null? (cdr expr))
                                '(0 0)
                                (pair-add
                                        (num-calls (cadr expr))
                                        (if (eval (cadr expr))
                                                (num-calls (cons 'and (caddr expr)))
                                                '(0 . 0)
                                        )
                                )
                        )
                )
    (else (cons (+ 1 (len expr)) 1))  ))
```

4. **Tail Recursion**

Which of the following functions are tail-recursive?

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

Not Tail Recursive

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

Tail Recursive

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

Tail Recursive

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

Tail Recursive

Fill in the following function definition so it's tail-recursive. It 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 lst seen res)
                (if (null? lst)
                        res
                        (helper
                                (cdr lst)
                                (car lst)
                                (if (= seen (car lst)) (car lst) res)
                        )
                )
        )
        (helper lst -1 #t)
)
```

5. **Where's Groot? (Fall 2014 Mock Final 4b)**

(b) **(8 pt)** 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 ((null? lst) '())
         ((list? (car lst))
           (append (deep-reverse (cdr lst))
                   (list (deep-reverse (car lst))) ))
         (else
           (append (deep-reverse (cdr lst))
                   (list (car lst)) )) ))
```