CS 61A Tail Recursion, Interpreters, Macros
Spring 2020 Guerrilla Section 4: November 16, 2019 Solutions

1 Tail Recursion

1.1 For the following procedures, determine whether or not they are tail recursive. If they are not, write why not and rewrite the function to be tail recursive on the right.

```
; Multiplies x by y
(define (mult x y)
  (if (= 0 y)
      0
      (+ x (mult x (- y 1)))))
```

```
(define (mult x y)
  (define (helper x y total)
    (if (= 0 y)
        total
        (helper x (- y 1) (+ total x))))
  (helper x y 0))
```

Not tail recursive: after evaluating the recursive call, we still need to apply '+', so evaluating the recursive call is not the last thing we do in the frame.

```
; Always evaluates to true
; assume n is positive
(define (true1 n)
  (if (= n 0)
      #t
      (and #t (true1 (- n 1)))))
```

Tail recursive: the recursive call to "true1" is the final sub-expression of the 'and' special form. Therefore, we will not need to perform any additional work after getting the result of the recursive call.

```
; Always evaluates to true
; assume n is positive
(define (true2 n)
  (if (= n 0)
      #t
      (or (true2 (- n 1)) #f)))
```

```
(define (true2 n)
  (if (= n 0)
```

```
      #t
      (true2 (- n 1))))
```

Not tail recursive: the recursive call to "true2" is not the final sub-expression of the 'or' special form. Even though it will always evaluate to 'true' and short-circuit, the interpreter does not take that into account when determining whether to evaluate it in a tail context or not.

```
; Returns true if x is in lst
(define (contains lst x)
  (cond
    ((null? lst)            #f)
    ((equal? (car lst) x)   #t)
    ((contains (cdr lst) x) #t)
    (else                   #f)))


(define (contains lst x)
  (cond
    ((null? lst)          #f)
    ((equal? (car lst) x) #t)
    (else                 (contains (cdr lst) x))))
```

Not tail recursive: the recursive call to "contains" is in a predicate sub-expression. That means we will have to evaluate another expression if it evaluates to true, so it is not the final thing we evaluate.

1.2 Tail recursively implement **sum-satisfied-k** which, given an input list **lst**, a predicate procedure **f** which takes in one argument, and an integer **k**, will return the sum of the first **k** elements that satisfy **f**. If there are not **k** such elements, return 0.

```
; Doctests
scm> (define lst `(1 2 3 4 5 6))
scm> (sum-satisfied-k lst even? 2)  ; 2 + 4
6
scm> (sum-satisfied-k lst (lambda (x) (= 0 (modulo x 3))) 10)
0
scm> (sum-satisfied-k lst (lambda (x) #t) 0)
0

(define (sum-satisfied-k lst f k)

    (define (sum-helper lst k total)
      (cond
        ((= 0 k)
         total)
        ((null? lst)
         0)
        ((f (car lst))
         (sum-helper (cdr lst) (- k 1) (+ total (car lst))))
        (else
         (sum-helper (cdr lst) k total))))

    (sum-helper lst k 0)

)
```

1.3 Tail-recursively implement **remove-range** which, given one input list **lst**, and two nonnegative integers **i** and **j**, returns a new list containing the elements of **lst** except the ones from index **i** to index **j**. You may assume $\mathbf{j} > \mathbf{i}$, and **j** is less than the length of the list. (Hint: you may want to use the built-in **append** function)

```
; Doctests
scm> (append '(1 2) '(3 4) '(5 6))
(1 2 3 4 5 6)
scm> (remove-range '(0 1 2 3 4) 1 3)
(0 4)

(define (remove-range lst i j)

    (define (remove-tail lst index so-far)
      (cond
        ((> index j)
         (append so-far lst))
```

```
    ((>= index i)
     (remove-tail (cdr lst) (+ index 1) so-far))
    (else
     (remove-tail (cdr lst)
                  (+ index 1)
                  (append so-far (list (car lst))))))
  (remove-tail lst 0 nil))
```

)

Check your understanding

- Why aren't all subexpression evaluations tail-recursive? For instance, why isn't the evaluation of `(+ 4 5)` as part of evaluating `(+ 1 (+ 2 3) (+ 4 5))` tail recursive, even though it's the last expression in the summation?

  In most cases, after evaluating a subexpression, we must take what it evaluates to and use it in its enclosing expression. For instance, when evaluating `(+ 1 (+ 2 3) (+ 4 5))`, after recursively evaluating `(+ 4 5)` to be 9, we must go back and add it to the other expressions in the summation, before returning 15. However, in tail recursion, we are able to state *before evaluating the subexpression* that we will just pass its evaluated value back as the evaluated value of the base expression. For instance, when evaluating `(and #t (f x))`, we know that *whatever* `(f x)` evaluates to, that will be what the `and` expression evaluates to as a whole. Thus, our call to `f` is in a tail context.

- Given a function `(f lst)` that acts over a list that has a single recursive call of the form `(f (cdr lst))`, what would be a general approach for rewriting it tail-recursively?

  There are a number of reasonable strategies - one useful technique is to define a helper function with signature `(helper curr rest)`, where `rest` stores the portion of the list to be iterated over, and `curr` contains a single quantity representing the accumulated result over the visited portion of the list. When `rest` becomes `nil`, then the helper function can return `curr`. In the body of the helper function, it can make a single tail call of the form `(helper new-curr (cdr rest))`, so it iterates over the list in a tail-recursive manner.

2 Interpreters

2.1 Determine the number of calls to **scheme_eval** and the number of calls to **scheme_apply** for the following expressions. Use the visualizer at code.cs61a.org if you're not sure how an expression is evaluated.

```
> (+ 1 2)
3
```

4 calls to eval:
1. Evaluate entire expression
2. Evaluate +
3. Evaluate 1
4. Evaluate 2

1 call to apply:
1. Apply + to 1 and 2

```
> (if 1 (+ 2 3) (/ 1 0))
5
```

6 calls to eval:
1. Evaluate entire expression
2. Evaluate predicate 1
3. Since 1 is true, evaluate the entire sub-expression (+ 2 3)
4. Evaluate +
5. Evaluate 2
6. Evaluate 3

1 call to apply:
1. Apply + to 2 and 3

```
> (or #f (and (+ 1 2) 'apple) (- 5 2))
apple
```

8 calls to eval:
1. Evaluate entire expression
2. Evaluate false
3. Evaluate entire sub-expression (and (+ 1 2) 'apple)
4. Evaluate entire sub-expression (+ 1 2)
5. Evaluate +
6. Evaluate 1
7. Evaluate 2
8. Evaluate 'apple

Since the and expression evaluates to true, we short circuit here.
1 call to apply:
1. Apply + to 2 and 3

```
> (define (add x y) (+ x y))
add
> (add (- 5 3) (or 0 2))
2
```

13 calls to eval:
1. Evaluate entire define expression
2. Evaluate entire (add ...) expression
3. Evaluate add operator
4. Evaluate entire (- 5 3) sub-expression
5. Evaluate -
6. Evaluate 5
7. Evaluate 3
8. Evaluate entire (or 0 2) sub-expression
9. Evaluate 0 (and short circuit, since 0 is truthy in Scheme
10. Evaluate (+ x y) (after applying add and entering the body of the add function)
11. Evaluate +
12. Evaluate x
13. Evaluate y

2 call to apply:
1. Apply - to 5 and 3
2. Apply + to -2 and 0

Check your understanding

- When a Scheme interpreter evaluates a combination of the form `(a b c d e)`, when does it evaluate `a`? Does it do so when `a` evaluates to a user-defined function? What about a builtin procedure? What if it is a keyword for a special form?

  If `a` evaluates to a user-defined function or a builtin procedure, it will be evaluated. For instance, when evaluating `(+ 1 2 3)`, there are 5 calls to `scheme_eval` - the overall expression, the three integers, and the `+` itself. However, when evaluating a special form, the keyword will not be evaluated, since it is treated specially by the interpreter. For instance, when evaluating `(and 1 2 3)`, there will only be 4 calls to `scheme_eval`, since the symbol `and` is a keyword and so will not be evaluated.

- What happens when we redefine a builtin procedure, like `#[+]`? For instance, if we run `(define + -)`, and then `(+ 1 2)`, what do we get? What about if we overwrite a keyword corresponding to a special form?

  When we redefine a builtin procedure, we will change its behavior, since it is evaluated just like a user-defined procedure. In the above example, after redefining `+`, `(+ 1 2)` will evaluate to -1! However, this is not the case for special forms, since their keywords are checked by the interpreter, not looked up in an environment. Thus, even if we run `(define and or)`, `(and #f #t)` will still evaluate to `#f`, since `and` will never be evaluated and looked up in the

current environment.

3 Macros

3.1 What will Scheme display? If you think it errors, write Error

```
> (define-macro (doierror) (/ 1 0))
```

```
doierror
```

```
> (doierror)
```

```
Error
```

```
> (define x 5)
```

```
x
```

```
>(define-macro (evaller y) (list (list 'lambda '(x) 'x) y))
```

```
evaller
```

```
> (evaller 2)
```

```
2
```

3.2 Consider a new special form, **when**, that has the following structure:

```
(when <condition> <expr1> <expr2> <expr3> ... )
```

If the condition is not false (a truthy expression), all the subsequent operands are evaluated in order and the value of the last expression is returned. Otherwise, the entire **when** expression evaluates to **okay**.

```
scm> (when (= 1 0)(/1 0) 'error)
okay
scm> (when (= 1 1) (print 6) (print 1) 'a)
6
1
a
```

Create this new special form using a macro. Recall that putting a dot before the last formal parameter allows you to pass any number of arguments to a procedure, a list of which will be bound to the parameter, similar to (*args) in Python.

```
; implement when without using quasiquotes
(define-macro (when condition . exprs)
```

```
    (list 'if ____________________________________________________________________________________)

; implement when using quasiquotes
(define-macro (when condition . exprs)
    `(if ____________________________________________________________________________________)


; without quasiquotes
(define-macro (when condition . exprs)
    (list 'if condition (cons 'begin exprs) ''okay))

; with quasiquotes
(define-macro (when condition . exprs)
    `(if ,condition ,(cons 'begin exprs) 'okay))
```