

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)))))
```

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

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

```
; 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)))
```

No, because the last clause isn't the last action of procedure.



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

⇒ This clause is not last action of a procedure.

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

- 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 (helper so-far num s)
```

```
    (cond ((null? s) and (< num k))
```

```
          0
          (= num k)
          so-far
```

```
          ((f (car s))
```

```
            (helper (+ (car s) so-far) (+ num 1) (cdr s)))
```

```
          (else (helper so-far num (cdr s))))))
```

(helper ? f k)

```
)
```

- 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 **j** > **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 (helper num so-far s)
```

```
    (cond ((= num j)
          so-far)
```

```
          (< num i)
```

```
            (helper (+ num 1) (append
```

```
              (else (helper (+ num 1) so-far (cdr s))))))
```

(remove-range lst i j)

)

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?
- 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?

① Because it's not the last action of this procedure even if it's last position.

② Use helper procedure to hold the so-far results.

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
```

eval apply print loop
REPL
 3

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

(pair 1 (pair (pair ' + ' (pair 2 (pair 3))) (pair (pair ' / ' (pair 1 (pair 0))))

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

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

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? *read if immediately*
- 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? *3*

we do nothing

3 Macros

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

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

doerror

```
> (doerror)
```

Error

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

x

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

5

```
> (evaller 2)
```

Error

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 condition (list 'begin exprs) 'okay))
```

```
; implement when using quasiquotes
```

```
(define-macro (when condition . exprs)
  `(if ,condition (begin ,exprs) okay))
```