Guerilla Section Week 6 Worksheet
Streams, Tail Recursion, Interpreters, Macros

Streams

1. Streams WWSD

```
scm> (define a (cons-stream 4 (cons-stream 6 (cons-stream 8 a))))

scm> (car a)


scm> (cdr a)


scm> (cdr-stream a)


scm> (define b (cons-stream 10 a))

scm> (cdr b)


scm> (cdr-stream b)


scm> (define c (cons-stream 3 (cons-stream 6)))

scm> (cdr-stream c)
```

What elements of a, b, and c have been evaluated thus far?

2. Write a function merge that takes in two sorted infinite streams and returns a new infinite stream containing all the elements from both streams, in sorted order.

```
(define (merge s1 s2)
```

3. Write a function `half_twos_factorial` that returns a new stream containing all of the factorials that contain the digit 2 divided by two. Your solution must use only the following functions, without defining any additional ones. Likewise, any lambda expressions should contain only calls to the following functions or built in functions.

```
; Returns a new Stream where each new value is the result of calling
; fn on the value in the stream s
(define (map-stream s fn)
      (if (null? s) s
            (cons-stream (fn (car s)) (map-stream (cdr-stream s)
fn))))

; Returns a new Stream containing all values in the stream s that
; satisfy the predicate fn
(define (filter-stream s fn)
      (cond ((null? s) s)
            ((fn (car s)) (cons-stream (car s) (filter-stream
(cdr-stream s) fn)))
            (else (filter-stream (cdr-stream s) fn))))


; Returns True if n contains the digit 2. False otherwise
(define (contains-two n)
      (cond ((= n 0) #f)
            ((= (remainder n 10) 2) #t)
            (else (contains-two (quotient n 10)))))

; Returns the factorial n
(define (factorial n)
      (if (= n 0) 1 (* n (factorial (- n 1)))))


; Returns a stream of factorials
(define (factorial-stream)
      (define (helper n)
            (cons-stream (factorial n) (helper (+ n 1))))
      (helper 1))
```

Fill in the skeleton below.

```
(define (half-twos-factorial)

    ___________________________________________

    ___________________________________________ )
```

Tail Recursion

1. For the following procedures, determine whether or not they are tail recursive. If they are not, write why they aren't and rewrite the function to be tail recursive to 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)))
```

2. Rewrite this function tail-recursively.

```
; Returns a list of pairs, the ith pair has item as its car and the
; ith element of lst as its cdr
(define (add-to-all item lst)
     (if (null? lst)
          lst
          (cons (cons item (car lst))
             (add-to-all item (cdr lst)))))
```

3. 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
```

Implement `sum-satisfied-k` tail recursively.

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

4. 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` in order, without the elements from index `i` to index `j` inclusive. 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, which returns the result of appending the items of all lists in order into a single well-formed list.)

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

(define (remove-range lst i j)
```

Now implement `remove-range` tail recursively.

```
(define (remove-range lst i j)
```

Interpreters

1. For the following questions, circle the number of calls to scheme_eval and the number of calls to scheme_apply:

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

Calls to `scheme_eval`:	1 | 3 | 4 | 6
Calls to `scheme_apply`:	1 | 2 | 3 | 4

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

Calls to `scheme_eval`:	1 | 3 | 4 | 6
Calls to `scheme_apply`:	1 | 2 | 3 | 4

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

Calls to `scheme_eval`:	6 | 8 | 9 | 10
Calls to `scheme_apply`:	1 | 2 | 3 | 4

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

Calls to `scheme_eval`:	12 | 13 | 14 | 15
Calls to `scheme_apply`:	1 | 2 | 3 | 4

Macros

Question 0

What will Scheme output? If you think it errors, write Error.

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

scm>(doierror)

scm> (define x 5)

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

scm> (evaller 2)
```

Question 1

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.

a) Fill in the skeleton below to implement this without using quasiquotes.

(define-macro (when condition . exprs)
(list 'if ____))

b) Now, implement the macro using quasiquotes.

(define-macro (when condition . exprs)
`(if ____))

Question 2

The goal of this question is to define a macro that represents a while loop. Since this is a difficult task we will break it into parts.

2a

Write tail-recursive factorial:

```
(define (fact n)









)
```

2b

Using the above problem to assist implementation, create the while macro. This macro will accept 4 arguments:

- initial-bindings: this will represent initialization values for variables in the loop
- condition: this will represent the condition which the while loop should continue to check to see if the loop should continue
- return: after the loop has ended this represents the value that should be returned

You may find the built-in map function useful for this problem:

```
scm > (map (lambda (x) (* 2 x)) '(1 2 3))
(2 4 6)
```

And here's an example of the while macro being used to calculate the factorial:

```
scm > (define (fact n)
      (while
            ((acc 1) (n n))
            (> n 0)
            ((* acc n) (- n 1))
           acc))
fact
scm> (fact 4)
```

24
Fill in the following macro definition:

```
(define-macro (while initial-bindings condition updates return)

    (define helper-vars
__________________________________________)

    (define initial-vals
_________________________________________)

    (list 'begin

        (list 'define (cons 'helper
_______________________________)

            `(if _________________________________________________

                 _________________________________________________


          _________________________________________________)


________________________________________________________))
```

CONGRATULATIONS!

You made it to the end of the worksheet! Great work.