# Guerilla Section Week 6 Worksheet

## Iterators & Generators, Streams, Tail Recursion, Interpreters

## Instructions

Form a group of 3-4. Start on Question 1. Check off with a lab assistant when everyone in your group understands how to solve the questions up to Checkpoint 1. Repeat for Checkpoint 2, 3, etc. **You're not allowed to move on from a checkpoint until you check off with a tutor.** You are allowed to use any and all resources at your disposal, including the interpreter, lecture notes and slides, discussion notes, and labs. You may consult the lab assistants, **but only after you have asked everyone else in your group. The purpose of this section is to have all the students working together to learn the material.**

## Iterators & Generators

1. Generator WWPD

```
>>> def g(n):
      while n > 0:
            if n % 2 == 0:
                  yield n
            else:
                  print('odd')
            n -= 1
>>> t = g(4)
>>> t


>>> next(t)


>>> n


>>> t = g(next(t) + 5)

>>> next(t)
```

2. Write a generator function `gen_inf` that returns a generator which yields all the numbers in the provided list one by one in an infinite loop. Write your solution to the right.

```
>>> t = gen_inf([3, 4, 5])
>>> next(t)
3
>>> next(t)
4
>>> next(t)
5
>>> next(t)
3
>>> next(t)
4
```

```
def gen_inf(lst):
```

3. Write a function `nested_gen` which, when given a nested list of iterables (including generators) `lst`, will return a generator that yields all elements nested within `lst` in order. Assume you have already implemented `is_iter`, which takes in one argument and returns `True` if the passed in value is an iterable and `False` if it is not.

```
def nested_gen(lst):
    '''
    >>> a = [1, 2, 3]
    >>> def g(lst):
    >>>   for i in lst:
    >>>         yield i
    >>> b = g([10, 11, 12])
    >>> c = g([b])
    >>> lst = [a, c, [[[2]]]]
    >>> list(nested_gen(lst))
    [1, 2, 3, 10, 11, 12, 2]
    '''

    ________________________

        if ___________________________________:

            ___________________________________

        else:

            ___________________________________
```

4. Write a function that, when given an iterable `lst`, returns a generator object. This generator should iterate over every element of `lst`, checking each element to see if it has been changed to a different value from when `lst` was originally passed into the generator function. If an element has been changed, the generator should yield it. If the length of `lst` is changed to a different value from when it was passed into the function, and `next` is called on the generator, the generator should stop iteration.

```
def mutated_gen(lst):
    '''
    >>> lst = [1, 2, 3, 4, 5]
    >>> gen = mutated_gen(lst)
    >>> lst[1] = 7
    >>> next(gen)
    7
    >>> lst[0] = 5
    >>> lst[2] = 3
    >>> lst[3] = 9
    >>> lst[4] = 2
    >>> next(gen)
    9
    >>> lst.append(6)
    >>> next(gen)
    StopIteration Exception
    '''
    ________________________________________
    ________________________________________
        curr = ________
        while ____________________:
            if ____________________________:
                break
            else:
                ___________________________________
                    yield ____________________
                ______________________________
    return ________________________
```

# STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section, and you have gotten checked off for **Check-in #1**

# 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)

    _____________________________________________

    _____________________________________________ )
```

# STOP!

Don’t proceed until everyone in your group has finished and understands all exercises

in this section, and you have gotten checked off for **Check-in #2**

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

Now implement `sum-satisfied-k` tail recursively.

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

# STOP!

Don’t proceed until everyone in your group has finished and understands all exercises in this section, and you have gotten checked off for **Check-in #3**

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 |

3a) In Discussion 11, we introduced the `Calculator` language, which is a Scheme-syntax language that currently includes only the four basic arithmetic operations: +, −, ∗, and /. In order to evaluate Calculator expressions, we've defined `calc_eval` and `calc_apply` as follows. Note that the basic arithmetic operations mentioned above are stored in the OPERATORS dictionary, which maps operator names to built-in functions.

```
def calc_eval(exp):
    """Evaluates a Calculator expression represented as a Pair. """
    if isinstance(exp, Pair):
        return calc_apply(calc_eval(exp.first),
                          list(exp.second.map(calc_eval)))
    elif exp in OPERATORS:
        return OPERATORS[exp]

    else: # Primitive expression
        return exp

def calc_apply(op, args):
    """Applies an operator to a Pair of arguments."""
    return op(*args)
```

For each of the following operations, select the function(s) that need to be modified in order to implement this new features in the *Calculator* language introduced in Discussion 11. Please justify your answer with 1-2 sentences.

The = operator. For example, `(= 5 5)` should evaluate to True.

calc_eval  calc_apply  Both  Neither

The `or` operator. For example, `(or (= 5 2) (= 2 2) (\ 1 0))` should evaluate to True.

calc_eval  calc_apply  Both  Neither

Creating and calling `lambda` functions (Assume `define` has been implemented.) For example: `(define square (lambda (x) (* x x))) (square 4)` should evaluate to 16.

calc_eval  calc_apply  Both  Neither

3b) Now, try implementing the `or` operator. You may assume that the conditional operator `<`, `>`, and `=` have already been implemented. To represent Scheme in Python, we are using Pair objects. A pair has two instance attributes: first and second. For a Pair to be a well-formed list, second is either a well-formed list or nil.

```
def calc_eval(exp):
     if isinstance(exp, Pair):

          ____________________________________________________________
          __

          ____________________________________________________________
          __

          ____________________________________________________________
          __

          ____________________________________________________________
          __
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else: # Primitive expression
        return exp

def eval_or(operands):
```

_____

_____

_____

_____

_____

_____

_____

_____

# STOP!

Don’t proceed until everyone in your group has finished and understands all exercises in this section, and you have gotten checked off for **Check-in #4**

## Challenge Problem

WWPD?

```
>>> def blue(purple, iter):
>>>         black = next(iter)
>>>         next(iter)
>>>         yield from purple[black]
>>> purple = [1, 2, 3, 4]
>>> red = iter(purple)
>>> orange = iter(red)
>>> yellow = iter(purple)
>>> purple[0], purple[1], purple[3] = 3, purple, list(purple)
>>> next(red)


>>> purple[2] = list(orange)
>>> next(red)


>>> green = blue(purple, yellow)
>>> purple[3][1] = list(green)
>>> purple[3][1][0]


>>> next(yellow)[1]


>>> purple[3][2]
```

```
>>> purple[2][2][1][3]
```

# CONGRATULATIONS!

You made it to the end of the worksheet! Great work :)