Guerilla Section Week 6 Worksheet

Iterators & Generators, Streams, Tail Recursion, Interpreters

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
Generator Object

>>> next(t)
4

>>> n
NameError: name 'n' is not defined

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

>>> next(t)
odd
6
```

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.

```
>>> 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):
  while True:
    for elem in lst:
      yield elem

def gen_inf(lst):
  while True:
     yield from iter(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]
     '''

     for elem in lst:
          if is_iter(elem):
               yield from nested_gen(elem)
          else:
               yield elem
     (solution using try / except instead of is_iter:)
     def nested_gen(lst):
          for elem in lst:
               try:
```

```
                iter(elem)
                yield from nested_gen(elem)
        except TypeError:
            yield elem
```

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.

Alternative wording: Write the mutated_gen function which given a list lst, returns a generator that only yields values that have been changed from their original value.
(or yields elements from the list that have been changed from their original value since lst was first passed in as an argument for mutated_gen)
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

    >>> lst2 = [1, 2, 3, 4, 5]
    >>> gen2 = mutated_gen(lst2)
    >>> lst2 = [2, 3, 4, 5, 6]
    >>> next(gen)
```

```
    StopIteration Exception #the list that the operand was
evaluated
                                to has not been changed.

    >>> lst3 = [1, 2, 3]
    >>> gen3 = mutated_gen(lst3)
    >>> lst3.pop()
    >>> next(gen)
    StopIteration Exception #the length of the list that was passed
                                in was changed
    >>> lst4 = [[1], 2 , 3]
    >>> gen4 = mutated_gen(lst4)
    >>> lst4[0] = [1]
    >>> next(gen)
    StopIteration Exception
    '''

    original = list(lst)
    def gen_maker(original, lst):
        curr = 0
        while curr < len(original):
            if len(original) != len(lst):
                break
            else:
                if original[curr] != lst[curr]:
                    yield lst[curr]
            curr += 1
    return gen_maker(original, lst)
```

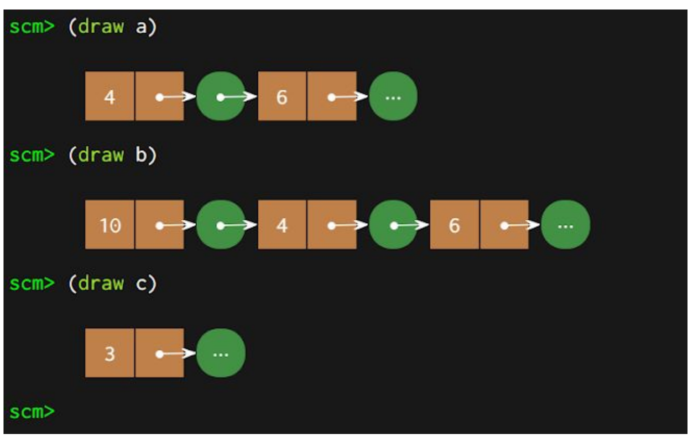
Streams

1. Streams WWSD

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

scm> (car a)

4

scm> (cdr a)

#[promise (not forced)]
```

```
scm> (cdr-stream a)

(6 . #[promise (not forced)])

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

scm> (cdr b)

#[promise (not forced)]

scm> (cdr-stream b)

(4 . #[promise (forced)])

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

scm> (cdr-stream c)

Error: too few operands in form
```

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)
        (if (null? s1)
              s2
              (if (<= (car s1) (car s2))
                     (cons-stream (car s1) (merge (cdr-stream s1) s2))
                     (cons-stream (car s2) (merge s1 (cdr-stream s2)))
              )
```

```
      )
)

; Alternate solution
(define (merge s1 s2)
      (if (null? s1)
            s2
            (if (<= (car s1) (car s2))
                  (cons-stream (car s1) (merge (cdr-stream s1) s2))
                  (merge s2 s1)
            )
      )
)
```

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)
      (map-stream (filter-stream (factorial-stream) contains-two)
                  (lambda (x) (quotient x 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)))))

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

Notn 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. (Review non-tail recursive factorial, the reason that is not tail recursive applies to this procedure)

```
; 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)))
```

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.

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

```
            (else (contains (cdr lst) x))))
```

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)))))

(define (add-to-all item lst)
      (define (helper item lst so-far)
            (if (null? lst)
            so-far
            (helper item (cdr lst) (append so-far (list (cons item
(car lst)))))))
      (helper item lst nil))
```

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)
      (cond ((= 0 k) 0)
            ((null? lst) 0)
            ((f (car lst)) (+ (car lst) (sum-satisfied-k (cdr lst) f
(- k 1))))
            (else (sum-satisfied-k (cdr lst) f k))
      )
)
```

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

```
(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))
```

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. For example, given the list `(0 1 2 3 4)`, with `i = 1` and `j = 3`, we would return the list `(0 4)`. 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)
      (define (helper lst index)
            (cond ((> index j) lst)
                  ((>= index i) (helper (cdr lst) (+ index 1)))
                  (else (cons (car lst) (helper (cdr lst) (+ index
1))))))
      (helper lst 0))
```

Now implement `remove-range` tail recursively.

```
(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)))
```

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

1. Evaluate entire expression
   a. Evaluate +
   b. Evaluate 1
   c. Evaluate 2
   d. **Apply** + to 1 and 2

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

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

1. Evaluate entire expression
   a. Evaluate predicate 1
   b. Since 1 is true, evaluate <if-true> sub-expression
      i. Evaluate +
      ii. Evaluate 2
      iii. Evaluate 3
      iv. **Apply** + to 2 and 3

Note that we never needed to evaluate the `if` because it's one of our special forms!

```
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

1. Evaluate entire expression
    a. Evaluate first sub-expression of or
    b. Evaluate second sub-expression of or
        i. Evaluate first sub-expression of and
            1. Evaluate +
            2. Evaluate 1
            3. Evaluate 2
            4. **Apply** + to 1 and 2
        ii. Evaluate ‘apple
        iii. Since the and expression evaluates to true, we short circuit

Note that we never needed to evaluate the `or` or the `and` because they are special forms!

```
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

1. Evaluate entire define expression
2. Evaluate call to add function
    a. Evaluate add function
    b. Evaluate first argument
        i. Evaluate -
        ii. Evaluate 5
        iii. Evaluate 3
        iv. **Apply** - to and 3
    c. Evaluate second argument (the or expression)
        i. Evaluate 0
        ii. Since 0 is #t, we short circuit
    d. **Apply** add function to 2 and 0 (enter body of the function)

i. Evaluate the body of the function
   1. Evaluate +
   2. Evaluate x to be 2
   3. Evaluate y to be 0
   4. **Apply** + to 2 and 0

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 Other

Justification: We are adding a new built-in operator, similar to the pre-defined +, -, * and / operators. Therefore, we will need to define an equality function. Then, we will need to update the `OPERATORS` dictionary to map the string `'='` to the new equality function.

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

`calc_eval` `calc_apply` Both Other

Justification: We must updated `calc_eval` in order to support the or operator's short circuiting.

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 Other

Justification: `lambda` is a special form, so it requires handling in `calc_eval`. Additionally, when user-defined lambda functions are called, we will need to create new frames when applying user-define lambda functions to their arguments. Therefore, we will need to update `calc_apply` to support creating new frames in these cases.

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):

        if exp.first == 'or':

            return eval_or(exp.second)

        else:

            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 eval_or(operands):

    curr = operands

    last = False

    while curr is not nil:

        last = calc_eval(curr.first)

        if last is True:

            return True
```

```
            curr = curr.second

        return last
```

Alternative solution:

```
def eval_or(operands):
        result = False
        while operands is not nil and result != True:
            result = calc_eval(operands.first)
            operands = operands.second
        return result
```

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)
3
>>> purple[2] = list(orange)
>>> next(red)
StopIteration Exception
>>> green = blue(purple, yellow)
>>> purple[3][1] = list(green)
>>> purple[3][1][0]
1
>>> next(yellow)[1]
3
>>> purple[3][2]
3
>>> purple[2][2][1][3]
4
```