Student ID: a1691807 **Name:** Yi-Ting, Hsieh

Question(1)

a) For our question1-(a), we also need to build other user-defined procedures to make things easier. Because our goal is to find the sum of the squares of all the even elements of mylist. So we need one primitive procedure for square, one for the sum and one for our filter function to check whether the input element is even number or not.

User-defined procedures:

(square x) : take one argument x, and return (* x x).

(sum x y) : take two arguments x and y, and return (+ x y).

(if-even? x) : take one argument x, if the x is odd number it will return 0(false), otherwise, it returns 1(true).

```
(define (q1a xs)
      (if (= (length xs) 0)
           0
           (cond
               ((= 0 (if-even? (car xs)))
                      (q1a (cdr xs)))
               ((= 1 (if-even? (car xs)))
                      (sum (square (car xs)) (q1a (cdr xs))))
            )
       )
  )
```

Above is the basic setup for this question. Right now we need to use recursive decomposition build up our procedure called q1a which will use above procedures for this question. Our q1a only takes one argument list xs, and it will return the sum of the square of even elements in list xs. Because we use recursursion for this question, so we need to find out base case, which is when the length of our input list xs is equal to zero, we return zero. And in our recursive call, in each call, we only handle one element, which is the leftmost element, in the list and check whether the element is even number or not, if so, we call the square primitive and put it into our sum procedures with q1a(rest of the list). And in every recursice call, we narrow down our list size by one.

Test case(mylist)	**expected**	**get**	**purpose:**
(list 1 2 3 5 8 9 10)	168	168	pass the required test case.
‘()	0	0	check it can handle empty list
(list 1 3 5)	0	0	check it can filter the odd elements
(list 2 4 6)	56	56	check it can sum up all the squares of even numbers
(list 1 2 3 4 5 6)	56	56	check it can behave as expected when our input contains both even and odd elements
(list 6 5 4 3 2 1)	56	56	check the order of our list will not affect our result

b) For our question1-(b), we need to use higher order abstractions to i our task. This time we could use some built-in procedures like map, foldr, and even?. Because we already have map and even? procedures, we only need to define our filter and foldr procedure. Basically I just follow our testbook and build a simple filter procedure and foldr. So my idea is quite simple, we go through all the elements in our list and check whether that element is even number, if so, we use square procedure and map procedure to store the square values and put them to a new list. And that's how we call the procedure: (map square (filter even? Mylist)). And right now, we need to sum up all the elements in our new list. We use foldr procedure do this sum up recursion. Basically foldr will go through all the elements in our new list, and if the size of our new list is nil, then it would add a zero and return back our sum up value. And in other recursive call(which is when the size of our list is bigger than zero), it will sum up the first element and recursive call itself with the rest of the list. By doing so, each time we will narrow down the size of the list by one.

User-defined procedures:

(filter predicate mylist): this will filter our input list, which is mylist

(foldr op z ls)

```
(define (q1b mylist)
        (foldr + 0 (map square (filter even? mylist)))
)
```

Test case(mylist)	**expected**	**get**	**purpose:**
(list 1 2 3 5 8 9 10)	168	168	pass the required test case.
'()	0	0	check it can handle empty list
(list 1 3 5)	0	0	check it can filter the odd elements
(list 2 4 6)	56	56	check it can sum up all the squares of even numbers
(list 1 2 3 4 5 6)	56	56	check it can behave as expected when our input contains both even and odd elements
(list 6 5 4 3 2 1)	56	56	check the order of our list will not affect our result

Question(2)

a) Because we already assume that the input lists are of equal length, we can easily make use of this property to do our recursive call. For our interleave procedure, the base case is when our lists are both empty, and we return a nil back. And in our recursive call, we either put our element from xs to our new list or put element from ys list. Because it has the same length, so during the whole recursive call, we will keep taking turns to put their elements into our new list. So I put a cond() to check which list we are going to handle for this call, if the length of our xs list and ys list are equal, then we put the front element of our current xs list to new list, otherwise we put the front element from our current ys list.

User defined procedures:

```
(define (interleave xs ys)
  (if (null? ys)
      (list)
      (cond ((= (length xs) (length ys)) (cons (car xs) (interleave (cdr xs) ys)))
            ((< (length xs) (length ys)) (cons (car ys) (interleave xs (cdr ys))))
      )
  )
)
```

Test case(xs	**ys)**	**expected**	**get**	**purpose:**
(list 1 2 3)	(list 4 5 6)	(1 4 2 5 3 6)	(1 4 2 5 3 6)	pass the required test case
‘()	‘()	()	()	check it can handle empty list
(list 4 5 6)	(list 1 2 3)	(4 1 5 2 6 3)	(4 1 5 2 6 3)	check whether it works when the size of input lists are odd
(list 1 2)	(list 3 4)	(1 3 2 4)	(1 3 2 4)	check whether it works when the size of input lists are even

b) Because for the second version, we don’t know what is the length of the input lists. It may be unequal length. So we need to check which list has the less length. And because we are going to implement our interleave procedure in recursion, we need to find out our base case and recurrence. Because we know if input xs and ys lists both have elements, then we will take one element from xs and one from ys and put them to our new list with the procedure itself of the rest of two lists. Which means each time we will narrow down length by one for both xs and ys lists. And the base case is when one of our list is nil or both our list are nil. If only one input list is nil, we put the rest of another list to our result list.

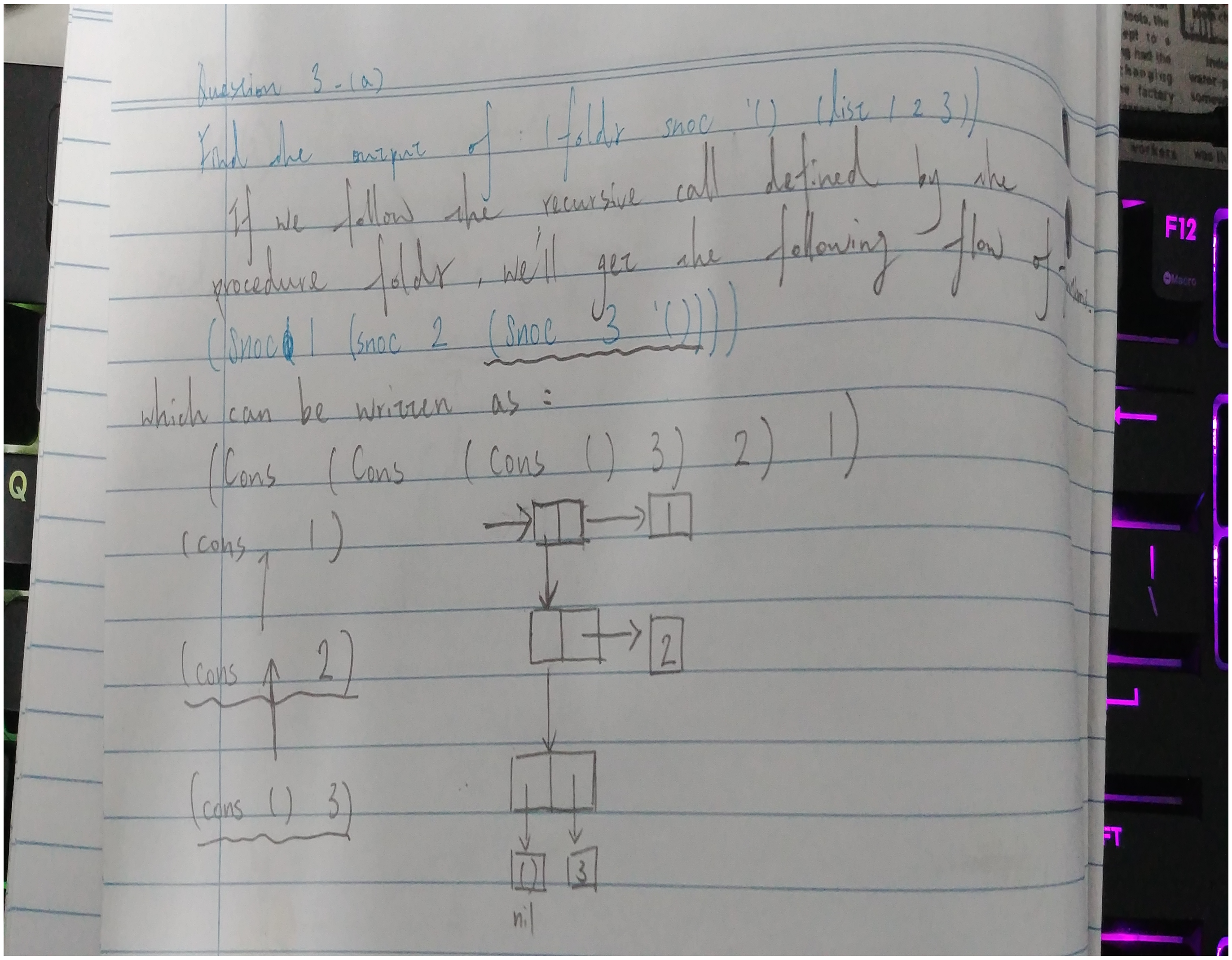
User defined procedures:

```
(define (interleave xs ys)
  (if (null? xs)
      (if (null? ys)
          (list)
          ys
      )
      (if (null? ys)
          xs
          (cons (car xs) (cons (car ys) (interleave (cdr xs) (cdr ys))))
      )
  )
)
```

Test case(xs	**ys)**	**expected**	**get**	**purpose:**
(list 1 2 3)	(list 4 5 6)	(1 4 2 5 3 6)	(1 4 2 5 3 6)	pass the required test case
‘()	‘()	()	()	check it can handle empty list
(list 4 5 6)	(list 1 2 3)	(4 1 5 2 6 3)	(4 1 5 2 6 3)	check whether it works when the size of input lists are odd
(list 1 2)	(list 3 4)	(1 3 2 4)	(1 3 2 4)	check whether it works when the size of input lists are even
(1 3 5)	(2)	(1 2 3 5)	(1 2 3 5)	check whether it works when the ys has smaller length than xs
(5 2)	(1 3 4 6)	(5 1 2 3 4 6)	(5 1 2 3 4 6)	check whether it works when the xs has smaller length than ys

Question(3)

a) The output of (foldr snoc ‘() (list 1 2 3)) is:
(((() . 3) . 2) . 1)



Question 3-(a)
Find the output of: (foldr snoc '() (list 1 2 3))
If we follow the recursive call defined by the procedure foldr, we'll get the following flow of [illegible]:
(snoc 1 (snoc 2 (snoc 3 '())))
which can be written as:
(Cons (Cons (Cons () 3) 2) 1)

(cons   1)

(cons   2)

(cons () 3)

b) Given the foldl, snoc and foldr procedures. We need to create another two procedures called f and g, such that rev1 and rev2 can both work correctly as list reversal procedures. Because by q3-(a), we know that (foldr snoc ‘() (list 1 2 3)) will create a list tree which is different from our expected reverse list (3 2 1). And the reason why is because during the recursive call in foldr, it will process the inner-most procedure first, which means it will find the base case and applies the procedure call, which is (op (3:last element in list) nil)). But if op is snoc, this will create an empty node links to these two nodes, which forms a list-tree structure. So how to avoid it? My idea is we could use append procedure. Basically it will take two lists and put one list to another tail, which means we will get a list back instead of the list-tree. So that is all my idea for procedure g; however, append requires two lists as arguments. But for our map procedure, it will only create one list, if we apply foldr or foldl with map procedures directly, we will have trouble about the incorrect argument during the recursive call. So for our f procedure, we need to define it as list. Basically it will help us create a list-tree structrue during the map process. By doing so, when our map process finish, we can use foldr and foldl and apply the g procedure to reverse our original list.

User-defined procedures:
(g ls1 ls2): append ls1 to ls2.

```
(define (g ls1 ls2)  (append ls2 ls1))
```

(define f list): in fact, our procedure f is list procedure, which will form a list for input.
(define (rev1 ls) (foldr g ‘() (map f ls))
(define (rev2 ls) (foldl g ‘() (map f ls))

Test case(mylist)	**expected**	**get(rev1)**	**(rev2)**	**purpose:**
(list 1 2 3)	(3 2 1)	(3 2 1)	(3 2 1)	pass the required test
(list 1 2 3 4)	(4 3 2 1)	(4 3 2 1)	(4 3 2 1)	check it works when the input list has even size
(list 1 2 3 4 5)	(5 4 3 2 1)	(5 4 3 2 1)	(5 4 3 2 1)	check it works when the input list has odd size
‘()	()	()	()	check it can work with nil

c)

Ans: (length (rev1 (enum-range 1 20000)) will take longer time to finish than the other two.

If we take a closer look at the definition of rev1 procedure, we will see this:
(define (rev1 ls) (foldr g ‘() (map f ls))).
Basically, it returns a procedure called foldr and execute the g procedure we defined ealier with two arguments, one is nil and the other is (map f (enum-range 1 20000)). If we take a closer look in our foldr procedure, the given definition is:

```
(define (foldr op z ls)
        (if (null? ls)
            z
            (op (car ls) (foldr op z (cdr ls)))))
```

As you can see, if the input list ls is nil, it will return z. But if ls is not nil, it will return the procedure op with two arguments, one is the leftmost element in the list ls, and the other is recursive call the foldr procedure with the rest of the list as argument. Because Scheme requires to evaluate all the arguments first, so it will try to get what is the value of the second argument, which is (foldr op z (cdr ls)). As we have some knowledge about how the computer execute our code, when the computer is reaching a new function call, it will allocate a new memory blocks for that function and changes the register to that area and executes the code then returns the value back. For this foldr procedure,let’s say right now the computer is already in the memory block which is allocated for the first foldr procedure, the computer will first allocate some memory space for the first op procedure and try to put the arguments in. But it will soon find out that there is another recursive call for foldr again in argument, so it has to allocate some memory space and find out what is that argument value again, and it will keep doing the same thing until it reaches the base case while the op procedure we called in the very beginning is waiting for all this very-long recursive call returns the result. And that is why it would take much longer time to get the result.