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

Question(1)
1a) For our question1-(a), we need to modify the make-account procedure so that it maintains and updates a transaction number every time when a transaction takes place. So we need a local counter to keep tracking the number of transactions and stores the value in the local variable—count. First, we need to define the counter procedure and the local variable—count inside our make-account. My idea is we define the local variable—count to store current state of the counter value. And how to implement our counter value? Well, we initial our counter to zero at first, and every time when the account is asked for transaction, it increase the counter by one. We could use “let” in our counter to establish an environment with a local environment. By using the lambda calculs and set! procedure in our counter, we could easily update our counter value and store it as local variable. And our counter only be called when a transaction is allowed to processed, which means if the user try to withdraw some money he doesn’t have, then this is not allowed and the transaction should not be processed, and our counter should remain the same value.

Pseudo code of our make-account procedures for our counter and count procedures:
```
(define (make-account balance)
  ;define our counoter to count how many transactions we call
  (define (counter initial)
    (let ((number initial))
      (lambda (transaction)
        (set! number (+ number transaction))
        number)))
  (define count (counter 0))
        ……
  )
```

While user can either deposit money to his/her accounts, or he/she can withdraw some money from it if the require amount is acceptable. In this scenario, we know we need another three procedures for our make-account. The dispatch procedure, which only has one argument, handles whether the user wants to deposit or withdraw the money. The second one is deposit procedure, which only has one argument, adds the amount to the balance. And the withdraw procedure, which has one argument, checks whether the balance is sufficient. And as we mentioned before, if the user asks for the transaction, once the transaction is allowed to process, our withdraw or deposit procedure should update the new balance and call our counter to update the transaction number.

Pseudo code of our make-account procedures for withdraw and deposit procedures:

```
(define (make-account balance)
  ……
(define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               (count 1)
               balance)
        "Insufficient funds"))
  ;define our deposit procedure
  (define (deposit amount)
    (set! balance (+ balance amount))
    (count 1)
    balance)
  )
```

1b) For our question1-(b), we need to modify our make-account so that it can respond to two additional messages: ‘balance will yield the current balance of the current balance of the account, and ‘transaction will yield the latest transaction number. Because we have a dispatch procedure, which will evaluate it’s argument— message and respond to the user. So if the user inputs ‘balance, then we just return the current balance of that account. And if the user inputs ‘transaction, we need to print out what is the current count value in our make-account. Be awared that when the user asks for ‘transaction and ‘balance, we do not need to update our count value, so when the user asks for ‘transaction, we need to call our count value but without updating it’s value.

Pseudo code of our make-account procedures for dispatch procedures:

```
(define (make-account balance)
  ……
 (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          ((eq? m 'transaction) (count 0))
          ((eq? m 'balance) balance)
          (else (error "Unknown request -- MAKE-ACCOUNT"
                       m))))
dispatch
 )
```

Test case	expected	get	purpose:
(define acc1 (make-account 100))			
(define acc2 (make-account 200))			
(acc1 'transaction)	0	0	check the initial transaction value
(acc2 'transaction)	0	0	should be zero
((acc1 'withdraw) 30)	70	70	check our withdraw works and update the balance and transaction counter
(acc1 'transaction)	1	1	check our transaction counter
(acc1 'balance)	70	70	check our balance is correct or not
(acc2 'transaction)	0	0	check when acc1 is handling the
(acc2 'balance)	200	200	request, it won't affect to acc2 values.
((acc2 'withdraw) 150)	50	50	check our acc2 can handle the request
(acc2 'transaction)	1	1	check our transaction counter works fine in acc2
(acc1 'transaction)	1	1	check when acc2 handle the request, it won't affect the acc1 values.
((acc1 'withdraw) 100)	"Insufficient funds"		check if the request is not valid, it should print out the warning message
(acc1 'transaction)	1	1	if the request is invalid, it shouldn't change our transaction counter
(acc1 'balance)	70	70	if the request is invalid, the object should has the same property as before
((acc1 'deposit) 100)	170	170	check user can deposit the money
(acc1 'transaction)	2	2	check it can update the transaction counter when user request for deposit
(acc1 'balance)	170	170	update the balance
(acc2 'transaction)	1	1	our acc2 should remain the same

Question(2)
2a) For our question2-(a), we need to complete the reorder procedure, which has two argument: one is order-stram and another is data-stream, to rearrange the items in our data-stream into the order specified by order-stream. Before we go any further, we need to define some procedures to help us first.

User defined procedures:

```
(define head car)
(define (tail stream) (force (cdr stream))
(define-syntax cons-stream
  (syntax-rules ()
    ((cons-stream x y)
     (cons x (delay y)))))

;define our stream-ref procedure
(define (stream-ref s n)
  (if (= n 0)
      (head s)
      (stream-ref (tail s) (- n 1))))

;define stream-null?
(define (stream-null? stream)
  (if (null? stream)
      #t
      #f))
```

Right now, we can focus on imeplementing our reorder procedure. The idea is when we are constructing our reorder stream, we use stream-ref to go through our data-stream and check which elements we want, specified by our order-stream, and put it to our reorder stream. And we could use the recursion inside our cons-stream instruction, so that it could go through our order-stream one by one and find out which element in our data-stream should be in our final stream. Once it reaches to the base case, which is when our order-stream or our data-stream is equal to empty stream, it will return the empty stream and stop.

Reorder procedure:

```
(define (reorder order-stream data-stream)
  (cond ((stream-null? order-stream) the-empty-stream)
        ((stream-null? data-stream) the-empty-stream)
        (else (cons-stream (stream-ref data-stream (- (head order-stream) 1))
              (reorder (tail order-stream) data-stream)))))
```

2b) We have just discussed about what our reorder procedure can do and how to use it. Now, let's evaluate what values may be returned by following instructions: (reorder integers data-stream1) and (reorder data-stream1 integers). Before we go any further, let's implement our integers procedure first. Basically it is an infinite streams started from one, and the next element is always the value of previous element increases by one.

Infinite procedure:

```
(define (integers-starting-from n)
  (cons-stream n (integers-starting-from (+ n 1))))

(define integers (integers-starting-from 1))
```

Because we already know that reorder rearranges the items in data-stream into the order specified by order-stream. So for the first case: (reorder integers data-stream1), because our integers stream is placed on the order-stream place, and we also know the integers is a stream like 1, 2, 3, 4…… . So our reorder procedure will go through the data-stream1 and put the first element in data-stream1 to our result stream, and it would put the second element in data-stream1 to the second place of our result stream, and so on. Basically, we will get another stream which is the same as our data-stream1.

For the second case: (reorder data-stream1 integers). This time, we need to reference our data-stream1 as the order-stream and integers as the data-stream. Basically, the reorder procedure will reference the first element in our data-stream1, for example, if our data-stream1 is (4, 13, 2, 8), and the first element is 4. It will find out what is the 4th element in the data-stream which is our integers this time. And we've already known the property of our integers, so our reorder procedure will put the 4th element in the integers, which is 4, to our result stream. And it will keep doing the same thing until it reaches the end of our data-stream1, so we will get the result stream as (4, 13, 2, 8), which is the same as our data-stream1.

2c) For question2-(c), we need to write a procedure print-first-n which takes two arguments, a stream s and a number n, and display the first n elements of the stream on one line separated by commas. Because we already define the procedure integers beforoe, we can focus on how to implement our print procedure. My idea is we could use recursive call to implement our print procedure that means if our input stream is not equal to null stream, then it will display the head of this stream and call the print-first-n with the new arguments--the tail of current stream and decreases the current n value by one. And if it reaches to our base case, which is when the input n is equal to zero, then we prints a newline for it. (Assume that the number n would always less than or equal to the length of our input stream s)

Print-first-n procedures:

```
(define (print-first-n s n)
  (cond ((= n 0) (newline))
           ((= n 1) (begin (display (head s))
                        (print-first-n the-empty-stream (- n 1))))
           (else  (begin (display (head s))
                      (display ", ")
                      (print-first-n (tail s) (- n 1))))))
```

2d) For question2-(d), we need to display the first seven numbers in the stream returned by (reorder (tail fibs) (tail fibs)). Because we already done the print-first-n procedure and reorder procedure, we can just implement our procedure fibs directly. Basically it's already defined in the lecture slides.

Fibs procedures and some other procedures:

```
(define (addL x y)
   (cons-stream (+ (head x) (head y))
             (addL (tail x) (tail y))))

(define fibs
  (cons-stream 1
        (cons-stream 1
              (addL (tail fibs) fibs))))
```

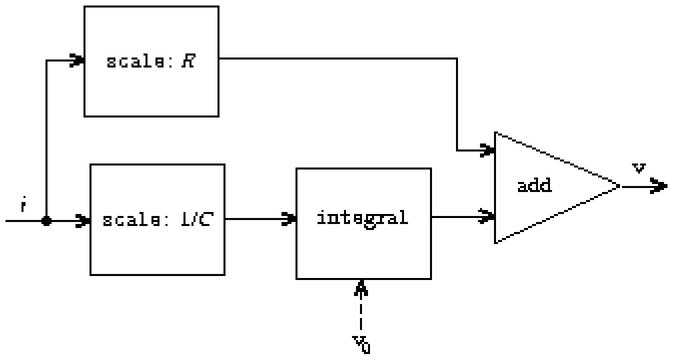
So the result for (print-first-n (reorder (tail fibs) (tail fibs)) 7) is:
   1, 2, 3, 8, 34, 377, 17711
Because by question-2(a), we already know that the reorder procedure will rearrange the items in data-stream into the order specified by order-stream. And this time, our data-stream and order-stream are all coming from the fibs stream. And we only want to print the first seven element of our stream after reordering, so we need to check what is the first seven elements in our order-stream, by calling (print-first-n (tail fibs) 7):
   1, 2, 3, 5, 8, 13, 21
Because this is coming from our order-stream, which means we need to find the first 21 elements in our data-stream and chose the elements in the position: 1, 2, 3, 5, 8, 13, 21 out and forms a new stream, which is our result.
By calling (print-first-n (tail fibs) 21):   (because we need at most 21 elements in our data-stream)
  1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711

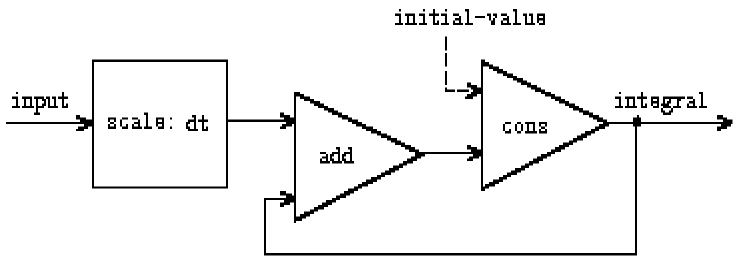
o	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
d	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181	6765	10946	17711

O:order stream   d:data-stream

And that's why we get our result:
        1, 2, 3, 8, 34, 377, 17711

Question(3)
3a) Basically we could use some of the procedures we've already done for question2. And from the SICP 3.73, we could see the diagram of this RC circuit would like the following diagram: And this is the design for our integral:





Integral Procedure:

```
;define the integral procedure
(define (integral integrand initial-value dt)
  (define int
    (cons-stream initial-value
                 (add-streams (scale-stream integrand dt)
                              int)))
  int)
```

RC Procedure:

```
#|define our RC procedure, which would take as inputs the values R, C and dt.
And it should return a procedure that takes as inputs a stream representing the
current i and an initial value for the capacitor voltage v_0 and produces as
output the stream of voltages.|#
(define (RC R C dt)
  (define (result input-stream initial-voltage)
    (add-streams (scale-stream input-stream R)
                 (integral (scale-stream input-stream (/ 1 C)) initial-voltage dt)))
  result)
```

So how to use our RC procedure. Basically we could define another procedure called RC1 which uses RC to mdel an RC circuit with R=5ohms, C=1 farad, and a 0.5 seconds time step by evaluating:

```
        (define RC1 (RC 5 1 0.5))
```

This defines RC1 as a procedure that takes a stream representing the time sequence of currents and an initial capacitor voltage and produces the output stream of voltages. Which means next time if we want to reference our RC1, we need to input the current stream and set up the initial capacitor voltage. It's like the following instruction:

```
        (RC1 current-stream initial-voltage)
```

And then, we could get the voltage stream of our circuit.

So, how to test it? Well, let's go back to our equation first.

$$v = v0 + (1/C)\int_{(0,t)} i\,dt + Ri$$

Let's print out our first six elements first:

0.005 0.0055 0.006 0.006500000000000001 0.007 0.0075

So, let's begin to calculate our voltage by putting the values into our equation:

The first voltage would be:

$$v = v0 + (1/C)\int_{(0,0)} i\,dt + Ri$$

$$= 0 + 0 + 5*0.001$$

$= 0.005(V)$ which is equviliant to our result

The second element:

$$v = v0 + (1/C)\int_{(0,0.5)} i\,dt + Ri$$

$$= 0 + (1/1)*(0.001)*(0.5-0) + 5*0.001$$

$= 0.0055(V)$ which is equviliant to our result

……

The sixth element:

$$v = v0 + (1/C)\int_{(0,2.5)} i\,dt + Ri$$

$$= 0 + (1/1)*(0.001)*(2.5-0) + 5*0.001$$

$= 0.0075(V)$ which is equviliant to our result

So our program can produce the voltage stream correctly as required.

3b)
User-defined test cases: (R=5, dt=0.5 secs)

```
;RC1 is our RC circuit with input C of 1F
(define RC1 (RC 5 1 0.5))
;RC2 is our RC circuit with input C of 1mF
(define RC2 (RC 5 0.001 0.5))
;RC3 is our RC circuit with input C of 10F
(define RC3 (RC 5 10 0.5))
```

(Assume that the current remains constant at 1mA)
(each one has twenty samples)
The output of our RC1(with 1F):

```
0.005 0.0055 0.006 0.006500000000000001 0.007 0.0075 0.008 0.0085 0.009000000000000001
0.009500000000000001 0.010000000000000002 0.010500000000000002
0.011000000000000003 0.011500000000000003 0.012000000000000004
0.012500000000000004 0.013000000000000005 0.013500000000000005
0.014000000000000005 0.014500000000000006
```

The output of our RC2(with 1mF):

```
0.005 0.505 1.005 1.505 2.005 2.505 3.005 3.505 4.005 4.505 5.005 5.505 6.005 6.505 7.005
7.505 8.005 8.505 9.005 9.505
```

The output of our RC3(with 10F):

```
0.005 0.00505 0.0051 0.00515 0.0052 0.00525 0.0053 0.005350000000000001 0.0054 0.00545
0.005500000000000005 0.00555 0.0056 0.005650000000000005 0.0057 0.00575
0.005800000000000005 0.00585 0.005900000000000001 0.00595
```

This is the samples I got for RC1, RC2 and RC3 under the same input current 1mA and the initial voltages are all set to zero.
As we can see from the above samples, for RC1, RC2 and RC3 have the same value at the beginning, which means the initial voltages are all 0.005(V). But we could see the voltage in the circuit with lower capacity, which is our RC2 with 1mF, increases much faster than RC1 and RC3. Every 0.5 sec, it increase around 0.5V than the previous stage. While the voltage in the circuit with highest capacity, which is our RC3 with 10F, increases much slowest than the rest. We could see it increases around 5*(10^(-5)) from the previous stage. And for RC1, RC2 and RC3, we could see the voltage increase with the same rates, RC1 increases 0.0005(V) every 0.5sec, RC2 increases 0.5(V) in 0.5 sec, and RC3 increases 0.00005(V) in 0.5 sec. Let's go back and evaluate our formula again:

$$v = v0 + (1/C)\int_{(0,t)} i\,dt + Ri$$

Because for our RC1, RC2 and RC3 circuit, out initial voltage, resistances, initial time and current are all the same. The only difference is they have different capacitance values. From the above equation, we can learn that the increase rates of the voltage in different RC circuit are porportional to the value of 1/C. Which means if the circuit has bigger capacitance value would have a smaller increase rate on voltage.

	RC1	RC2	RC3
C	1F	1mF = 0.001F	10F
1/C	1	1/0.001=1000	1/10=0.1
ΔVoltage	0.0005	0.5	0.00005

We could see from the above table:

$$1/C_{(RC1)} : 1/C_{(RC2)} : 1/C_{(RC3)} = 1 : 1000 : 0.1$$

is equivalent to

$$\Delta_{Voltage(RC1)} : \Delta_{Voltage(RC2)} : \Delta_{Voltage(RC3)} = 0.0005 : 0.5 : 0.00005 = 1 : 1000 : 0.1$$

This result is just the same as our expectation.

3c) For this question, we need to use the circuit we've already defined for the previous question, but fix the capacitance at 1F and analyze what happens when the current changes over time. Because we are asked to generate a stream of samples with the property that ten samples at 1mA, then ten at zero, then another ten at 1mA, and so on. So how to generate the stream can has ten samples at 1mA and 10 samples at zero? Because we will use scale-stream to scale our current values, so we only need to consider how to generate a stream with 10 samples at 1A and 10 samlpes at zero. Basically we could write an interleave procedure, which takes two stream and one required number, produces the stream with the required property.

User-defined test cases: (R=5, dt=0.5 secs)

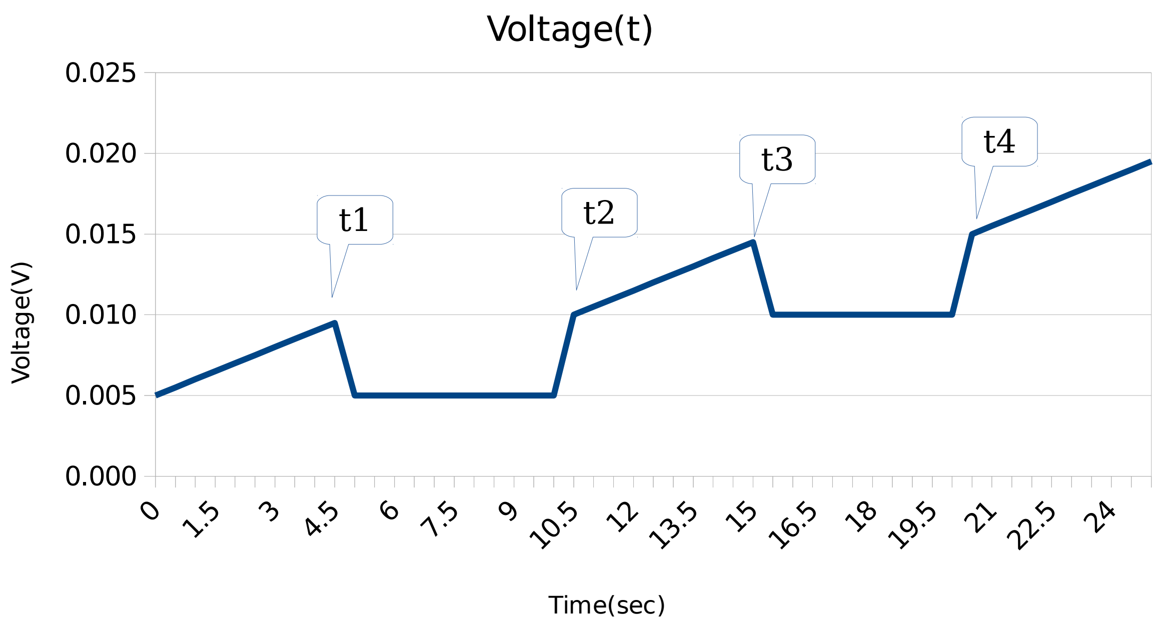
```
;define our zeros stream
(define zeros (cons-stream 0 zeros))
;define our stream-interleave10 procedure
;We only need to generate the total of 50 samples.
(define (stream-interleave10 stream-ones stream-zeros n)
  (cond ((= n 0)
         '())
        ((= n 10)
         (cons-stream-zeros 10 n stream-ones stream-zeros))
        ((= n 20)
           (cons-stream-zeros 10 n stream-ones stream-zeros))
    (else  (cons-stream (head stream-ones) (stream-interleave10 (tail stream-ones)
stream-zeros (- n 1))))))
;define cons-stream-zeros for passing the zeros stream
(define (cons-stream-zeros z n stream-ones stream-zeros)
  (if (= z 0)
      (cons-stream (head stream-ones) (stream-interleave10 (tail stream-ones)
stream-zeros (- n 1)))
  (cons-stream (head stream-zeros) (cons-stream-zeros (- z 1) n stream-ones (tail
stream-zeros)))))
;Define our current sample for q3-(c)
;because we only need 50 samples at total, which means there would only has
;30 samples come from ones stream
(define q3-current (stream-interleave10 ones zeros 30))
```

Basically, our q3-current has the stream which has 10 samples with 1A and 10 samples with 0A, and the total samples are up to fifty. Let's test on our RC circuit we've just defined earlier by calling following instruction:

```
(print-first-n (RC1 (scale-stream q3-current 0.001) 0) 50)
```

This is the result we get:

0	0.005	12.5	0.012000000000000004
0.5	0.0055	13	0.012500000000000004
1	0.006	13.5	0.013000000000000005
1.5	0.006500000000000001	14	0.013500000000000005
2	0.007	14.5	0.014000000000000005
2.5	0.0075	15	0.014500000000000006
3	0.008	15.5	0.010000000000000005
3.5	0.0085	16	0.010000000000000005
4	0.009000000000000001	16.5	0.010000000000000005
4.5	0.009500000000000001	17	0.010000000000000005
5	0.005000000000000001	17.5	0.010000000000000005
5.5	0.005000000000000001	18	0.010000000000000005
6	0.005000000000000001	18.5	0.010000000000000005
6.5	0.005000000000000001	19	0.010000000000000005
7	0.005000000000000001	19.5	0.010000000000000005
7.5	0.005000000000000001	20	0.010000000000000005
8	0.005000000000000001	20.5	0.015000000000000006
8.5	0.005000000000000001	21	0.015500000000000007
9	0.005000000000000001	21.5	0.016000000000000007
9.5	0.005000000000000001	22	0.016500000000000008
10	0.005000000000000001	22.5	0.017000000000000008
10.5	0.010000000000000002	23	0.01750000000000001
11	0.010500000000000002	23.5	0.01800000000000001
11.5	0.011000000000000003	24	0.01850000000000001
12	0.011500000000000003	24.5	0.01900000000000001
12.5	0.012000000000000004	25	0.01950000000000001



From above diagram, we could see that our result streams increase at the same rate at first 5 seconds, but when our RC circuit receives the zero current stream, the voltage falls back to the initial voltage value. After ten zero streams input, our RC circuit receives the current with 1mA and starts to increases the voltage value again before the zero stream comes in, which is around 0.01(V). The interesting thing is it has the same increase rate as before, which is because the increase rate of the voltage depends on the fixed capacitance, so the increase rate will not change. After 5 seconds, when the RC circuit receives the zero current again, the circuit voltage falls back from 0.02(V) to the 0.01(V) and stays for another 5 seconds, and the circuit receives 1mA, it back to 0.02(V) and increases the voltage with the same increase rate.

So, why this happens? Let's go back to our original formula:

$$v = v0 + (1/C)\int_{(0,t)} i dt + Ri$$

Whenever our RC circuit gets a zero current, our equation would get:

$$v = v0 + (1/C)0*t + R*0 = v0$$

This is why we get the previous value(v0) when our circuit receives the zero current stream. And after 5 seconds, the RC circuit receives the 1mA current stream again, but this time, it's slightly different.

$$\begin{aligned} v' &= v0 + (1/C)\int_{(0,t)} i dt + Ri \\ &= v0 + (1/C)\int_{(0,t1)} i dt + (1/C)\int_{(t1,t2)} 0 dt + (1/C)\int_{(t2,t)} i dt + R*1 \\ &= v(t1) + 0 + (1/C)\int_{(t2,t)} i dt + R*1 \end{aligned}$$

This time, our v(t1)is the value of the state before our circuit receives the zero current streams, which is 0.01(V). And because we didn't change our capacitance, so the increase rate remains the same. Then, once our circuit receives the zero current stream again, the voltage falls back from 0.015(V), which is our next v(t3), to 0.010(V) (V(t2)) and stay at V(t2) for another 5 seconds. Then once the circuit gets the 1mA current stream again, it starts from $v(t3)$ and increases the voltages.