COMP1811 – Scheme Project Report

Name	Hasan Ahmati	SID	001259023

1. Software Design

Briefly describe the design of your coursework software and the strategy you took for solving it. – e.g. did you choose either recursion or high-order programming and why…

For this coursework, I choose for each task different ways of solving the problems in scheme. I have used recursion, meta-programming, high-order programming and set-union as well.

Before starting to solve it I created a mini-plan on how I will complete this coursework after having a look at all the tasks. In most of the places that I have used recursion is because I had already done something in an other task so I used recursion to directly take the function that I needed and in this way, I would reduce reusing the code and time as well.

To start with the first task, I used recursion as I saw that task as the typical recursion case exercises where to reuse data we use recursion. I firstly check if the list is empty, if condition is true it will return an empty list. If the condition is false it will return a pair that takes the first key-value pair as the first element and then using recursion in the same list I add the remaining of the pair and append it to the cons.

I have massively used recursion on the third file to be repaired, sn.social-network.rkt . In this file I use recursion for a second function that I create called network_func as I need to return the result of the first element to this new function and the first function contains the list of the friends of a given user. This will be very needed for the other tasks as in the second task of the same file, I have to take the friends of 2 users given, so instead of creating the same code for each member I reuse the data of sn-ff-for function by using recursion to call this function inside the function that we are coding. I will need these users as I need to check the common friends and create a new common friends list that comes after filtering the 2 lists(that contain the friends of each user) and return the new list that contain the friends that are in both lists.

The list of friends will be used in all the tasks of this file. Even in the third count, where I have to return a list of pairs. These elements of these pairs will be (user id/name . number of their friends). I can get the number of their friends easily by just using recursion to call this function(sn-ff-for) and get the length of the list of friends. In the fourth and fifth tasks, in which the difference is that one requests the user with maximal number of friends and the other with the minimal number of friends, I need to use recursion to call sn-ff-for function to get the length of users and I also need to use recursion to call functions that I created to get the maximum and minimum digits of values.

Other than recursion I have also used meta programming, when I use its functions such as apply that I use in the second task of the file sn-utils.rkt. Here I use apply to convert a list of lists into a single one. Also in the third task I convert the strings into symbols and symbols can be considered part of meta-programming so I can say that I have used meta-programming.

The function 'map' of high-order programming is used almost everywhere in my code and was maybe the most used function by me. Other than map, I have used 'apply' to combine and manipulate the other functions [such as map or max/min(build-in functions)]. Another function that I used and helped me a lot is 'filter'. For instance at the task that requires to return that common friends of 2 users, I used filter to check the two lists of friends of these users and get the common values that the both lists have. I also used filter at the friendliest, unfriendliest tasks where I use filter to filter the list and return a list with elements that are true for the lambda condition.

The last function that I am going to describe why I used it is set-union . In the function sn-add-frndshp I have used set-union to combine two lists, the existing list the contains the friends of user 1 and the list of user 2. By using set-union it will combine the elements and return a new list and also remove duplicates.

Note: To use some functions, especially let, let* or even to get some more information about combinations such as car(car)) = caar I have used different sources from internet to get more information how to use them (I got idea how to use these functions in scheme and I did not include any part of code from them in my code). The main resources that I got the information from are:

https://franz.com/support/documentation/ansicl.94/dictentr/carcdrca.htm

https://docs.racket-lang.org/reference/let.html

Also, for certain methods and functions I got help from the tutor Mohammad Al-Antary and also a cousin Hajredin Daku. I have cited them in code and what was the help that I got from them in the comments of codes.

2. Code Listing

Provide a complete listing of the entire Scheme code you developed. Make sure your code is well commented, specially stressing informally the contracts for parameters on every symbol you may define. The code listed here must match that uploaded to Moodle. Please copy and paste the actual code – no screenshots please! Make it easy for the tutor to read. Marks WILL BE DEDUCTED if screenshots are included. Add explanatory narrative if necessary – though your in-code comments should be clear enough.

FUNCTION 1.1 ...

```
; using recursion to call the function itself
  (define (sn-list->dict lst)
    (cond ((empty? lst) '()) ;firstly we check if the list is empty(it does have any key-value pair to add) it will return an empty list
          (else (cons (car lst) (sn-list->dict (cdr lst)))))) ; else so if it not empty, it will create a pair that takes the first key-value pair from the list using car
  ; then using recursion we call the function itself and add the remaining of the pairs using cdr and appends them to the cons that was already created

```

FUNCTION1.2 ...

```
;using meta-programming(apply)
  (define (sn-dict-ks-vs ks vs) ;the function takes the list keys/ks and the list values/vs as input.
    (apply map cons (list ks vs))) ; list is used to create a single list that contains keys and values.
  ;map will apply cons to the corresponding elements from keys and values.
  ;map will return a list of lists. To convert the list of cons into a single list, we use apply to combine the inner lists into a single one
```

FUNCTION1.3 ...

```
;using meta-programming(symbols)
  (define (sn-line->entry ln) ;it takes one argument that is ln which is the list of entries
    (let* ;by using let* it allows me to define variables that are depended on each other.
        ([lst (string-split ln)] ;split the input string into a list of substrings
         [symbols (map string->symbol lst)]) ;converting every substrings into symbols using map
      (cons (first symbols)(rest symbols)))) ;using cons creating the list with the first symbol as key and the other symbols as value.
```

FUNCTION2.1 ...

```
;; Easy (+0.5)
  (define sn-empty
    empty)
```

FUNCTION2.2 ...

```
;using high-order programming (map)
  (define (sn-users graph)  ;; takes the dictionary graph as input
  (map car graph))  ;;map will extract the first element of the dictionary graph which are the keys/so they're the users
```

FUNCTION2.3 …

```
 ;; Hard
;; [(k,v)] u -> [(k,v)] | (u,{})
;Got help from a cousin Hajredin Daku,student in masters for cyber security. He told me to use (,user . ()) format to create the dotted pair.

(define (sn-add-user graph user) ;function takes two arguments graph(social network) and user. In this function we're going to add a user to the social network.
  ;the first condition is that if the graph/network is empty it will return  a new graph that has the user a key and an empty list as values.
  (cond ((empty? graph) `((,user . ()))) ;the (,user . ()) format: "," makes sure that user is taken as a variable. "." creates a dotted pair.
        ((equal? user (caar graph)) graph) ;now we check if the user added is the same with the first user(caar) already existing in the graph. If #t it will return the graph unchanged
        (else (cons (cons user '()) graph)))) ;if both conditions are false it will add the new user as key into the dictionary graph.

```

FUNCTION2.4 …

```
;; Hard
;; [(k,v)]|(u1,f1)|(u2,f2) ->
;;  [(k,v)] | (u1,f1+{f2}) | (u2,f2+{f1})
;using set-union
(define (sn-add-frndshp graph u1 u2)
  ; usig let,we create a recursive loop with 2 variables:graph and new_list and their initial values are graph and empty list
  (let loop ((graph graph) (new_list '())) ;until the graph list is empty the let loop will continue the call itself
    (cond
      [(empty? graph) new_list] ;if the network is empty it will return the new_list that contains the updated friendship list for the two users
      [(equal? u1 (car (car graph))) ;here we check if the first user is present it the first element of the graph.
      ;the first car check the first element of the network,second one the first element of the list of the user.
       (let ((new-friends (set-union (cdr (car graph)) (list u2))))
         ;if the user1 is found,the let will create a new list of friends(new-friends) by combining the existing list of user1 "(cdr (car graph))" with a new list containing user2
         ;using set-union that will combine the lists.
         (loop (cdr graph) (cons (cons u1 new-friends) new_list)))] ;recursively call the let to update the new_list
      [(equal? u2 (car (car graph))) ;in a similar way we check if user 2 is present in the nework
       (let ((new-friends (set-union (cdr (car graph)) (list u1)))) ;if it is it will do the same thing as it did with user1
         (loop (cdr graph) (cons (cons u2 new-friends) new_list)))]
2.51          [else (loop (cdr graph) (cons (car graph) new_list))]))) ;if both users are not found in the network,it will move on with the next entry in the network
```

FUNCTION3.1 ...

```
2.52   ;; Easy
2.53   ;; [(k,v)]| (u,vu) -> vu
2.54   ;using recursion(with the second function)
2.55   (define (sn-ff-for graph u1) ;defines a function with 2 parameters, graph(social network) and u1(user)
2.56     (define (network_func graph) ;defines a helper function called network_func that takes the graph/network only
2.57       (cond
2.58         [(empty? graph) '()] ;if the network is empty,returns an empty list
2.59         [(equal? u1 (caar graph)) (cdar graph)]  ;if the first element is equal to u1 then return the second element which is the list of the friends of this user
2.60         [else (network_func (cdr graph))])) ;if not it will continue looping for the rest of the network
2.61     (network_func graph)) ;calls the helper function as ss-ff-for needs to return the result of the 'network_func' function after it finished executing.
```

FUNCTION3.2 ...

```
2.62   ;; Medium
2.63   ;; [(k,v)]|(u1,f1)|(u2,f2) ->
2.64   ;; f2 & f3
2.65   ;using high-order programming(filter) and recursion to call the sn-ff-for function
2.66   (define (sn-cmn-frnds-btwn graph u1 u2) ;defining a function that takes three arguments.
2.67    ;Graph that contains the social network,u1 that is user1 and u2 user 2
2.68     (let ;defines fr1 and fr2
2.69         ((fr1 (sn-ff-for graph u1)) ;gets the friends of user 1
2.70          (fr2 (sn-ff-for graph u2))) ;gets the friends of user2
2.71       (filter (lambda (fr) (member fr fr1))fr2))) ;it returns a new list that has only the element of fr2 that are in fr1.
2.72   ;filter will return the list of the common friends it will apply lambda to each friend of user 1
2.73   ;lambda checks if friend("fr") is present in the list of friends of the user given("fr1").
2.74   ;if the friend is present in both lists(filter checks this) it will be returned as a common friend.
```

FUNCTION3.3 ...

```
2.75   ;; Hard
2.76   ;using recursion(calling sn-ff-for function again as we need the list of friends of a user for the count)
2.77   (define (sn-frnd-cnt graph)
2.78     (let ((users (map car graph))) ; taking the users, same as at sn-users
      (map (lambda (user) (cons user (length (sn-ff-for graph user))))users)))
  ; map will map each user in the users list to a new list of pair with this format (user name . number of friends), cons will create this pair
  ; lambda takes user as an argument and will return the user and the length of friends of that user
  ; using recursion we call the sn-ff-for function that has the friends of the user,and we take the length of the list of friends.
```

Function3.4 ...

```
;; pre: length > 0
  ;using build in functions(max), meta-programming(apply), high-order programming(filter)
  (define (max-ks extr_vs lst) ;function max-key takes 2 arguments, extr_vs will extraxt a value from each element in the lst
    (let
        ((max-vs (apply max (map extr_vs lst)))) ;we apply the max function(a build-in function) to the list of values that we take from "(map extr_vs lst)"
      (car (filter (lambda (x) (= (extr_vs x) max-vs)) lst)))) ;lambda will take "x" as argument and will check if this x is equal to the max-vs(the maximal value)
  ; filter will filter the list and return a list with the elements that are #t at the lambda condition
  ; car will take the first element of the list that was filtered and return this first element

  ;using recursion(to call the method max-ks and sn-ff-for) and elements of high-order programming(map)
  (define (sn-frndlst-user graph) ;the argument that this function takes is graph/network
    (let
        ((users (map car graph))) ;this way we take the users same as we did at sn-users
      (max-ks (lambda (user) (length (sn-ff-for graph user)))users)))
  ;here the function max-ks takes the lambda function of user(which returns the length of the list of friends), and the users list
  ;we use recursion as we created the list of friends of an user at the sn-ff-for function
  ;the max-ks function will return the friendliest user.
```

Function3.5 ...

```
       ;; pre: length > 0
         ;using build in functions(min), meta-programming(apply), high-order programming(filter)
         ;in the same way as at the maximum but here we take the minimal value

         (define (min-ks extr_vs lst)
           (let
               ((min-vs (apply min (map extr_vs lst)))) ;as we want the min number of friends we apply the build-in function min instead of max that we used above
```

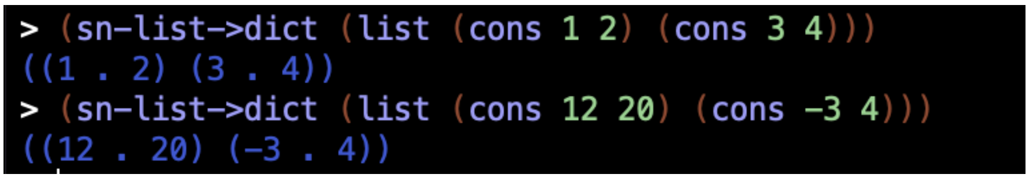
```
            (car (filter (lambda (x) (= (extr_vs x) min-vs)) lst))))

        ;using recursion(to call the method min-ks and sn-ff-for) and elements of high-order programming(map)
        (define (sn-unfrndlst-user graph)
          (let
              ((users (map car graph)))
            (min-ks (lambda (user) (length (sn-ff-for graph user)))
                    users)))
```

…

3. Results – Output Obtained

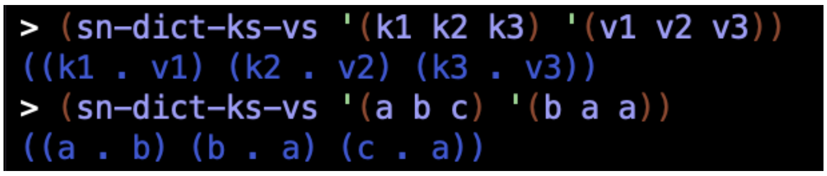
Provide screenshots that demonstrate the results generated by running your code. That is show the output obtained in the REPL when calling your functions. Alternatively, you may simply cut and paste from the REPL.

TASK-1.1



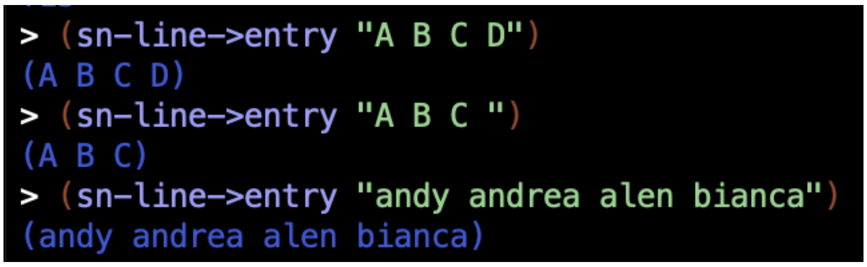
```
> (sn-list->dict (list (cons 1 2) (cons 3 4)))
((1 . 2) (3 . 4))
> (sn-list->dict (list (cons 12 20) (cons -3 4)))
((12 . 20) (-3 . 4))
```

TASK-1.2



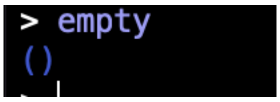
```
> (sn-dict-ks-vs '(k1 k2 k3) '(v1 v2 v3))
((k1 . v1) (k2 . v2) (k3 . v3))
> (sn-dict-ks-vs '(a b c) '(b a a))
((a . b) (b . a) (c . a))
```

TASK-1.3



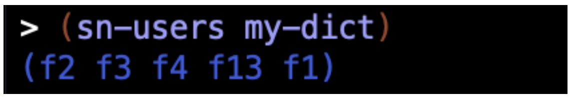
```
> (sn-line->entry "A B C D")
(A B C D)
> (sn-line->entry "A B C ")
(A B C)
> (sn-line->entry "andy andrea alen bianca")
(andy andrea alen bianca)
```

TASK-2 .1



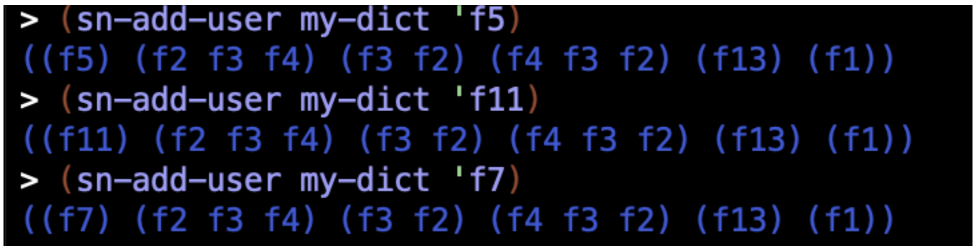
```
> empty
()
```

3.1 TASK-2 .2



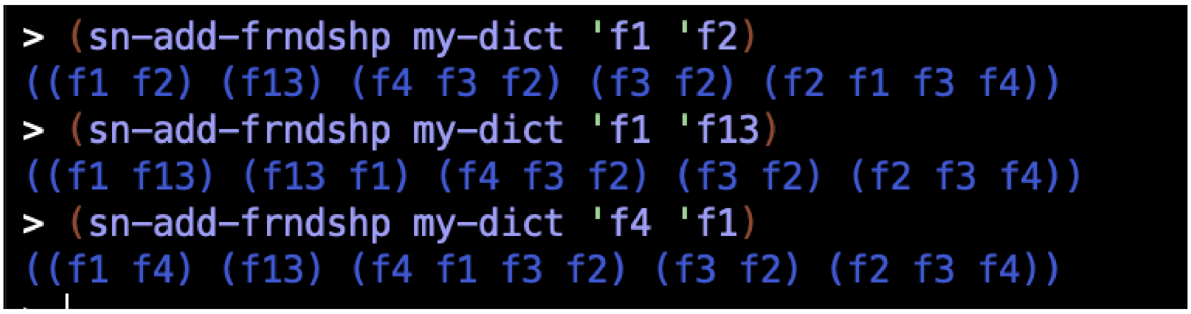
```
> (sn-users my-dict)
(f2 f3 f4 f13 f1)
```

TASK-2 .3



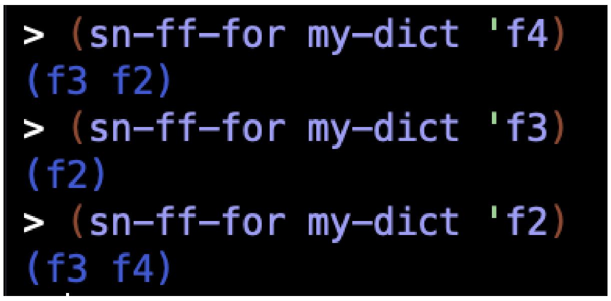
```
> (sn-add-user my-dict 'f5)
((f5) (f2 f3 f4) (f3 f2) (f4 f3 f2) (f13) (f1))
> (sn-add-user my-dict 'f11)
((f11) (f2 f3 f4) (f3 f2) (f4 f3 f2) (f13) (f1))
> (sn-add-user my-dict 'f7)
((f7) (f2 f3 f4) (f3 f2) (f4 f3 f2) (f13) (f1))
```

3.2 TASK-2 .4



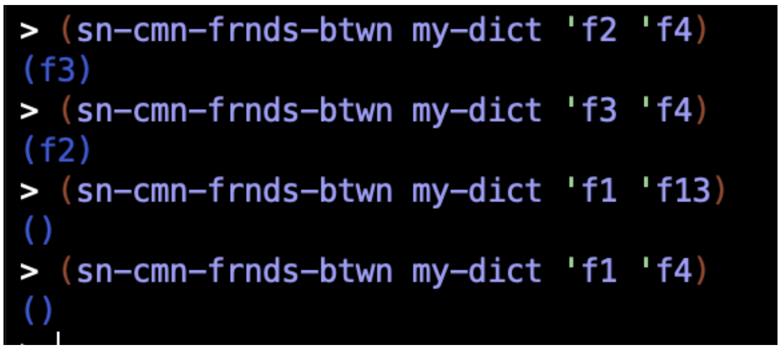
```
> (sn-add-frndshp my-dict 'f1 'f2)
((f1 f2) (f13) (f4 f3 f2) (f3 f2) (f2 f1 f3 f4))
> (sn-add-frndshp my-dict 'f1 'f13)
((f1 f13) (f13 f1) (f4 f3 f2) (f3 f2) (f2 f3 f4))
> (sn-add-frndshp my-dict 'f4 'f1)
((f1 f4) (f13) (f4 f1 f3 f2) (f3 f2) (f2 f3 f4))
```

TASK-3 .1



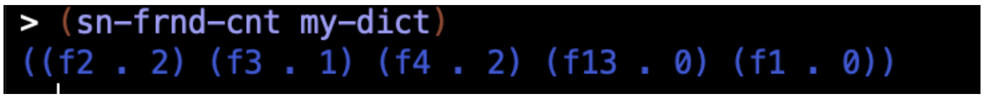
```
> (sn-ff-for my-dict 'f4)
(f3 f2)
> (sn-ff-for my-dict 'f3)
(f2)
> (sn-ff-for my-dict 'f2)
(f3 f4)
```

TASK-3 .2



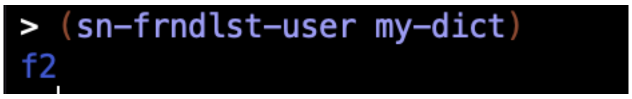
```
> (sn-cmn-frnds-btwn my-dict 'f2 'f4)
(f3)
> (sn-cmn-frnds-btwn my-dict 'f3 'f4)
(f2)
> (sn-cmn-frnds-btwn my-dict 'f1 'f13)
()
> (sn-cmn-frnds-btwn my-dict 'f1 'f4)
()
```

TASK-3 .3

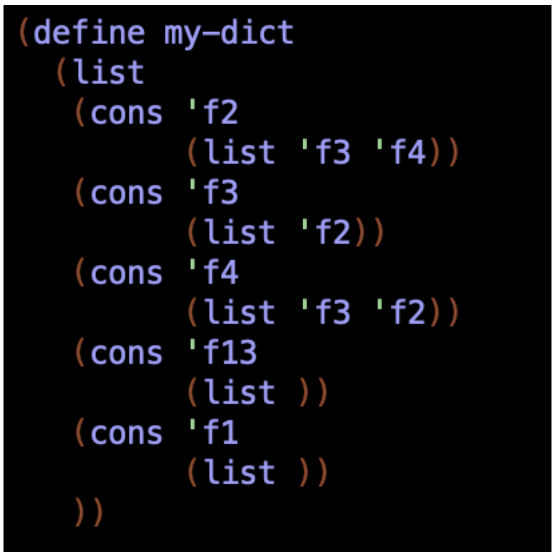


```
> (sn-frnd-cnt my-dict)
((f2 . 2) (f3 . 1) (f4 . 2) (f13 . 0) (f1 . 0))
```

TASK-3 .4



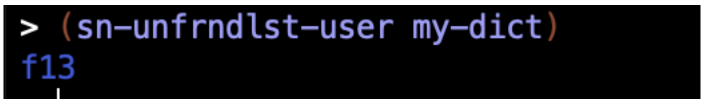
```
> (sn-frndlst-user my-dict)
f2
```



```
(define my-dict
  (list
   (cons 'f2
         (list 'f3 'f4))
   (cons 'f3
         (list 'f2))
   (cons 'f4
         (list 'f3 'f2))
   (cons 'f13
         (list ))
   (cons 'f1
         (list ))
   ))
```

There are 2 friendliest users but in the code I use the condition to show the first element of the list using car so that's why f2 is the output

TASK-3 .5



```
> (sn-unfrndlst-user my-dict)
f13
```

Same happens here as there are f13 and f1 with 0 members but in the code I only want one of them to be as an output.

…

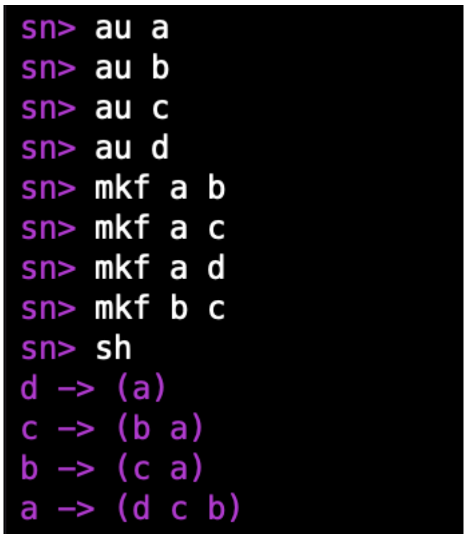
4. TESTING

Provide a test plan covering all of your functions and the results of applying the tests.

To test each task and the code that I gave for the solution I firstly used testing.rkt file. In this file there is a network given and also the functions to run them. For the tasks such as 3.1 , 3.2 etc I tested the code with different inputs as for example in the task 3.2 there are no common friends between f13 and f1 so I wanted to see if the output will be correct or not. Also at task 3.1,I tested one time giving one user ,for instance f3 and it shows that f3 is friend with f2,then I try with f2 to see if this output is correct or no. I get the conclusion that the output is correct as at the list of friends of f2 in the output,f3 is there. If we check each feature one by one we can see that the output is correct according to the request of the task.

Task 1.1 requests as an output to create a list of pairs for a given list of entries and as we can see at the output it gives a list of pairs for the input given. Task 1.2 for a given list of values and keys it should return a dictionary(a list of keys with their corresponding values). As we can see when we test it at testing.rkt the output is (key. value) for all the keys and values given. In this way we test all other features. My plan of testing was to test each feature after each implementation of code at testing.rkt file first. The testing varies for each task as there can be only one test at task 2.2 for instance ,where the only output should be the list of users, or for friendlies and unfriendliest users, and the testing will be different for tasks 3.1 or 3.2 as I mentioned above as there can be more than one test for different inputs and we can receive different outputs.

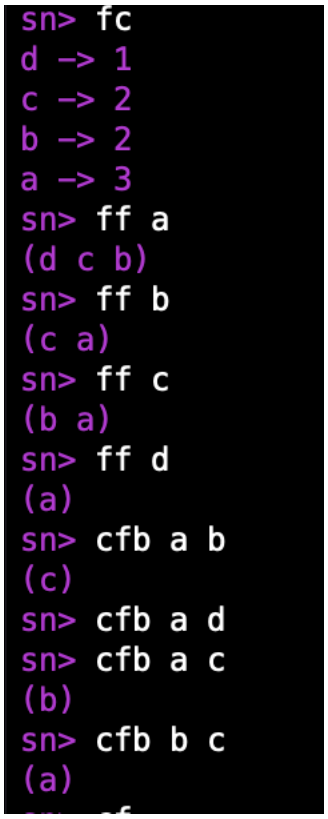
Another way of testing is the file sn-app.rkt. This file is provided at the files for the coursework and here I create a social network and then test all the features just like at the demo provided in the coursework.
Firstly I create the social network



```
sn> au a
sn> au b
sn> au c
sn> au d
sn> mkf a b
sn> mkf a c
sn> mkf a d
sn> mkf b c
sn> sh
d -> (a)
c -> (b a)
b -> (c a)
a -> (d c b)
```

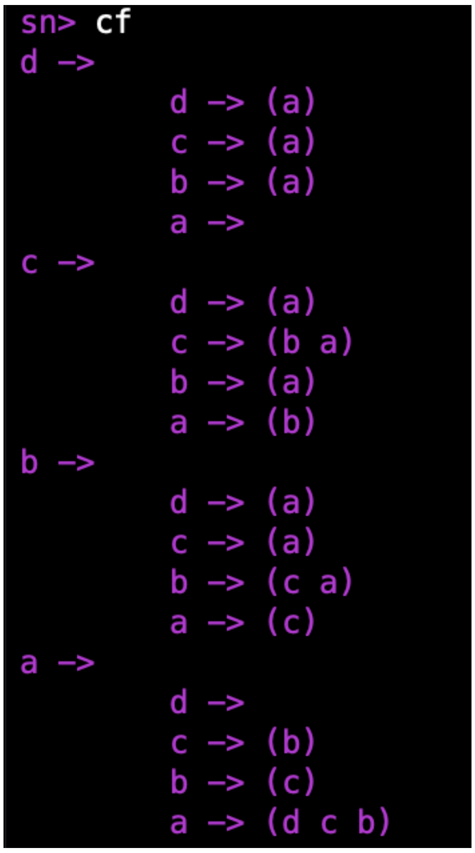
In this picture we add 4 users and make friends in between them. The social network is shown in the end using ‘sh’. As it is shown in the picture the network is correct and each member has been allocated with their correct friend.

In the next picture, I tested if the friend count and friends for methods work. As it is shown the friend count for each user is correct, same as the friend for each member. In the same picture, the common friends between/ cfb method is tested. We can check at the photo above that the common friend for a and c is b as they both are friends with b. Also I tested for users that don't have any common friends in between them and it doesn't show anything but passes to the next input.

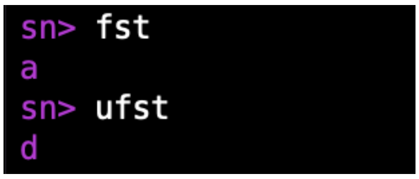


```
sn> fc
d -> 1
c -> 2
b -> 2
a -> 3
sn> ff a
(d c b)
sn> ff b
(c a)
sn> ff c
(b a)
sn> ff d
(a)
sn> cfb a b
(c)
sn> cfb a d
sn> cfb a c
(b)
sn> cfb b c
(a)
```

The third test that I did for the sn-app.rkt file is the common friends all(cf) method. The picture below is the result of this method. It shows the common friends between all the users. For example user a: has no common friends with user d, with c has b as a common friends and the same has c as a common friend between a and b. At itself it will show the list of friends that the user has.


```
sn> cf
d ->
        d -> (a)
        c -> (a)
        b -> (a)
        a ->
c ->
        d -> (a)
        c -> (b a)
        b -> (a)
        a -> (b)
b ->
        d -> (a)
        c -> (a)
        b -> (c a)
        a -> (c)
a ->
        d ->
        c -> (b)
        b -> (c)
        a -> (d c b)
```

The other features are friendliest and unfriendliest users.


```
sn> fst
a
sn> ufst
d
```

In this picture we can see that the friendlies user is a and the unfriendliest user is d. If we check the number of friends, we can see that a has 3 friends, b and c have 2 friends and d has only 1 friend. In this way we prove that the output and code is correct.

In conclusion, comparing the outputs that I received at sn-app.rkt file with the output received at the demo at the coursework specification, I realised that they are the same outputs (the users that are used as input at the demo are different from mine,but the structure and output is the same) and my code for the corresponding tasks is correct and valid.

5. Evaluation

*Evaluate your implementation and discuss what you would do if you had more time to work on the code. Critically reflect on the following point and write **300-400 words overall**.*

Points for reflection:

- what went well and what went less well?

What went well I would say was time-management. Even why this task was easier than the one that we completed on the first coursework using python as the problem is given in a simple way to solve and testing and some other stuff are already given, we had less time and I used this time in a more efficient and less stressful way.

What went less well was probably the convert from string to symbols as I found myself blocked as it was the beginning of the coursework, but with the help of the tutor I did very well later on.

- what did you learn from your experience? (not just about Scheme, but about programming in general, project development and time management, etc.)

what I leant from this experience was that there are many different ways to solve a problem given. Also I learnt that we can re-use data in an effective way using recursion. Another thing that I learnt is that at functional programming is easier to repair damaged file and in less time using functional programming.

- how would a similar task be completed differently?

For solving different tasks, in most of them I used recursion and I could've used high-order programming, but I wanted to reuse data and I found recursion as a better and more effective way to solve these tasks

- what can you carry forward to future development projects?

I have some things that I can carry for future projects from this coursework:

1.Experience: I spent more time using scheme functions and approaches to solve a real-life problem. This made me feel more confident using functional programming.

2.Time-management: I learnt how to manage time in a effective way

3.Reusing data: I experimented a lot more with recursion and usually it was the first approach that came in my mind when I was looking at each task. I used recursion a lot and it helped me to reuse the data that I needed and also saved me time that I could use redoing the same thing.

4.Logical thinking: There were some complex tasks in this coursework that logical thinking was needed. As programming is a lot of logic, this coursework helped me to improve logical thinking for programming.