## COMP1811 – Scheme Project Report

| **Name** | **Luis Nieto Hueso** | **SID** | **00119959** |
| --- | --- | --- | --- |

# Software Design

For this coursework, I combined recursion and higher-order programming. In order to identify the most and least friendly users, higher-order programming was employed more frequently. The ultimate output is a composite of the results from the smaller units, and recursion aids in the breakdown of huge issues into smaller units that can be solved using an algorithm. Control flow or algorithms that can be contained in procedures can be abstracted away using higher-order programming.

# 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.*

## TASK-1 util

## Function 1

*;; Returns a dictionary as a list of key-value pairs*

*;; sn-dict-ks-vs : list-of-any list-of-any -> list-of-(cons any any)*

*;; Given two lists `keys` and `values`, constructs a list of key-value pairs*

*;; by pairing corresponding elements from the two lists.*

  (define (sn-dict-ks-vs keys values)

    (map cons keys values))

The function sn-dict-ks-vs, and has two parameters, in the code above.

keys: An element list of any kind. The keys that will be used in the resulting dictionary are contained in this list.

values: An element list of any kind. The values that will be utilized in the resulting dictionary are contained in this list. The function pairs corresponding elements from the two input lists of keys and values to produce a list of pairs (key-value pairs). Also, is added the define that as the name is said define the name of the function to after be call

## Function 2

*;; Returns a dictionary entry from a string, where the first word is the key and the rest are the value*

*;; sn-line->entry : string -> (cons symbol list-of-string)*

*;; Given a string `ln`, returns a dictionary entry where the first word is the key*

*;; and the rest are the value.*

  (define (sn-line->entry ln)

  (let ((words (string-split ln)))

    (cons (string->symbol (first words)) (rest words))))

The method sn-line->entry, which is defined in the code above, has one parameter:

ln: The entry to be transformed into a dictionary entry is contained in a string.

The function gives back a dictionary entry (a pair), with the first word of the input string, ln, serving as the key and the other words as the value. To do this, we use the string-split function to first create a list of words from the input string ln, and the cons function to turn that list of words into a pair. Using the string->symbol function, we turn the first word—the key—into a symbol while keeping the remaining words—the value—as a list.). Also, is added the define that as the name is said define the name of the function to after be call

## Function 3

*;; Returns the input list of key-value pairs*

*;; sn-list->dict : list-of-(cons any any) -> list-of-(cons any any)*

*;; Given a list of key-value pairs `entries`, returns the same list.*

  (define (sn-list->dict entries)

    entries)

)

We have defined the function sn-list->dict in the code above, and it only accepts one parameter:

a list of key-value pairs called entries.

The function basically returns the key-value pair input list. When we already have a list of key-value pairs but need to generate a dictionary, we can use this function. Also, is added the define that as the name is said define the name of the function to after be call

## TASK-2 GRAPH

## Function 1

*;; sn-empty : -> empty*

*;; Returns an empty graph.*

  (define sn-empty empty)

The function sn-empty, which has no parameters, is defined in the code above.

The empty Scheme value, which is used to represent an empty list, is all that the function returns. This can be used to represent an empty graph in the context of a graph. Also, is added the define that as the name is said define the name of the function to after be call

## Function 2

*;;This function takes a graph as input and returns a list of all the users in the graph.*

*;; sn-users : (dict-of symbol any) -> list-of-symbol*

*;; Given a graph `graph`, returns a list of all the users in the graph.*

  (define (sn-users graph)

    (dict-keys graph))

The function called sn-users in the code above, and it only accepts one parameter:

a dictionary where the values stand in for the users' friends and the keys for the users themselves.

A list of all the users in the input graph is returned by the function. The dict-keys function, which pulls all the keys—i.e., users—from the input dictionary graph, allows us to accomplish this. Also, is added the define that as the name is said define the name of the function to after be call

## Function 3

*;;This function takes a graph and a user as input, and adds the user to the graph if they are not already in it.*

*;; sn-add-user : (dict-of symbol any) symbol -> (dict-of symbol any)*

  (define (sn-add-user graph user)

    (if (dict-ref graph user #f)

        graph

        (dict-set graph user empty)))

In the above code, we have defined a function sn-add-user that takes in two parameters:

graph: A dictionary where the keys represent users and the values represent the user's friends.

user: A symbol representing the user to be added to the graph.

The function adds the input user to the input graph if the user is not already in the graph. We achieve this by checking if the user is already present in the graph using dict-ref function. If the user is already present in the graph, the function simply returns the input graph. If the user is not present in the graph, the function adds the user to the graph using dict-set function, with an empty list as the value. Also, is added the define that as the name is said define the name of the function to after be call.

## Function 4

*;; This function takes a graph and two users as input, and adds a friendship between the two users. It does this by adding each user to the other's friend list in the graph.*

*;; sn-add-frndshp : (dict-of symbol (set-of symbol)) symbol symbol -> (dict-of symbol (set-of symbol))*

*;; Given a graph `graph` and two users `u1` and `u2`, adds a friendship between the two users.*

*;; It does this by adding each user to the other's friend list in the graph.*

  (define (sn-add-frndshp graph u1 u2)

    (let ([f1 (dict-ref graph u1 empty)]

          [f2 (dict-ref graph u2 empty)])

      (dict-set graph

                u1

                (set-add f1 u2))

      (dict-set graph

                u2

                (set-add f2 u1))))

the function sn-add-frndshp, which has three parameters, in the code above.

a dictionary where the values stand in for the users' friends and the keys for the users themselves.

u1: A symbol designating a user who will be added to the friend list of another user.

u2: A picture of the person who will be added to the first user's list of friends.

By adding each user to the friend list of the other, the function creates a friendship between users u1 and u2 in the input graph. To do this, we first fetch each user's buddy list using the dict-ref function, and then we add the other user to the friend list using the set-add function. After that, we use the dict-set method to update the input graph.

## Function 5

*;;This function takes a graph as input and returns #t if the graph is consistent, i.e., if for each user in the graph, all of their friends are also in the graph.*

*;;The function uses a recursive helper function to check each user in the graph and their friends.*

*;; all of their friends are also in the graph.*

*;; sn-consistent : (dict-of symbol (set-of symbol)) -> boolean*

  (define (sn-consistent graph)

    (define (consistent-helper [to-check (dict-keys graph)])

      (cond

        [(empty? to-check) #t]

        [else

         (let ([current-user (first to-check)]

               [remaining-users (rest to-check)])

           (define (friend-list-incomplete? user)

             (ormap (lambda (friend)

                      (and (dict-ref graph user #f)

                           (set-member? (dict-ref graph friend #f) user)))

                    (dict-ref graph user empty)))

           (if (friend-list-incomplete? current-user)

               #f

               (consistent-helper remaining-users)))]))

    (consistent-helper (dict-keys graph))))

The function sn-consistent in the code above, and it only takes one parameter:

a dictionary where the values stand in for the users' friends and the keys for the users themselves.

When a user and all of their friends are present in the graph, the function verifies whether the input graph is consistent. This is done using the recursive helper function consistent-helper, which takes a list of users to verify as an input parameter. The helper function checks whether every user has a complete friend list, or if every friend is represented in the graph. In the event that any user's friend list is incomplete, the method returns #f, signifying that the graph is inconsistent. The function with the remaining users calls Consistent-helper recursively to check that all friend lists are complete. After there are no more users remaining to verify, the function returns #t, demonstrating the consistency of the graph.

## TASK-3 SOCIAL NETWORKING

Tienes que editar esto para que no parezca to robotico

## Function 1

*;; This function takes a graph and a user as input, and returns a list of the user's friends.*

*;; sn-ff-for : (dict-of symbol (set-of symbol)) symbol -> list-of-symbol*

(define (sn-ff-for graph user)

  (let ([friend-set (dict-ref graph user #f)])

    (if friend-set

        (set->list friend-set)

        '())))

## I have defined the function sn-ff-for, which has two parameters, in the code above.

## a dictionary where the values stand in for the users' friends and the keys for the users themselves.

## user: A representational symbol for the person whose friends are being returned.

## The function retrieves the set of friends from the input graph using the dict-ref function before returning a list of the user's friends. If the set of friends contains no empty elements, the method uses the set->list function to turn the set into a list. The function produces an empty list if the user is not included in the graph or if the set of friends is empty.

## Function 2

*;; sn-cmn-frnds-btwn : (dict-of symbol (set-of symbol)) symbol symbol -> list-of-symbol*

*;;This function takes a graph and two users as input, and returns a list of their common friends.*

 (define (sn-cmn-frnds-btwn graph user1 user2)

  (let ([friend-set1 (dict-ref graph user1 #f)]

        [friend-set2 (dict-ref graph user2 #f)])

    (if (and friend-set1 friend-set2)

        (set->list (set-intersect friend-set1 friend-set2))

        '())))

## I have defined a function called sn-cmn-frnds-btwn in the code above that accepts three parameters:

## a dictionary where the values stand in for the users' friends and the keys for the users themselves.

## user1: A representation of a single user.

## user2: A representation of the opposing user.

## The function retrieves the set of friends for each user from the input graph using the dict-ref function before returning a list of the friends that users 1 and 2 have in common. If neither set of friends is empty, the function uses the set-intersect function to compute their intersection and uses the set->list function to return the result as a list. The function produces an empty list if either set of friends is empty or if either user1 or user2 is not present in the graph.

## Function 3

*;; sn-frnd-cnt : (dict-of symbol (set-of symbol)) -> list-of-pairs*

*;; This function takes a graph as input, and returns a list of users and their friend counts.*

(define (sn-frnd-cnt graph)

  (map (lambda (user)

         (cons user (set-count (dict-ref graph user empty))))

       (dict-keys graph)))

## I have defined the function sn-frnd-cnt in the code above, but it just requires one parameter:

## a dictionary where the values stand in for the users' friends and the keys for the users themselves.

## The function first maps the input graph's keys using the map function before returning a list of users and the number of friends they have. The function builds a pair of the user and the number of their friends for each user in the graph by applying the set-count function to the friends set for the user that was retrieved using the dict-ref function. The function's result is a list that contains all of the pairs.

## Function 4

*;; max-by : (a -> number) (list-of a) -> a*

*;; Given a function `func` and a list `lst`, returns the element of `lst` with the maximum value of `func`*

*;; return the element with the maximum*

  (define (max-by func lst)

  (if (null? lst)

      (error "empty list")

      (foldl (lambda (a b)

               (if (> (func a) (func b)) a b))

             (first lst)

             (rest lst))))

I have defined the function max-by, which has two parameters, in the code above.

func: A function that accepts an input list element and outputs a compareable number.

List of components to be compared with func, denoted by lst.

By first determining whether lst is empty, the function returns the element of lst with the highest value of func. The foldl function is applied to the remaining elements of the list, starting with the first one, if lst is not empty. The foldl function accepts a function that compares two elements by applying a function called func to them and then returning the one with the highest value of func. The function's output is the resultant element.

## Function 5

*;; sn-frndlst-user : (dict-of symbol (set-of symbol)) -> symbol*

*;;This function takes a graph as input, and returns the user with the most friends.*

(define (sn-frndlst-user graph)

  (let ([users (dict-keys graph)])

    (let\* ([counts (map (lambda (user)

                          (cons user (set-count (dict-ref graph user empty))))

                        users)])

      (max-by (lambda (x) (cdr x)) counts))))

I have defined the function sn-frndlst-user in the code above, and it only accepts one parameter:

graph: a dictionary where the values stand in for the users' friends and the keys for the users themselves.

The function first creates a list of pairs consisting of each user and the number of their friends by mapping the input graph's keys using the map function. This creates the user with the most friends. The function then uses the max-by function to determine the maximum count and returns the user corresponding to that count. The max-by function accepts two inputs: the input list itself and a function that retrieves the value to be compared from each element of the input list. In this instance, a lambda function is utilized to retrieve the count value from each pair, and the list of pairs generated earlier.

## Function 6

*;; min-by : (a -> number) (list-of a) -> a*

*;; Given a function `func` and a list `lst`, returns the element of `lst` with the minimum value of `func`.*

*;; return the element with the  minimum*

  (define (min-by func lst)

  (if (null? lst)

      (error "empty list")

      (foldl (lambda (a b)

               (if (< (func a) (func b)) a b))

             (first lst)

             (rest lst))))

## I have defined the function min-by, which has two parameters, in the code above.

## func: A function that accepts an input list element and outputs a compareable number.

## List of components to be compared with func, denoted by lst.

## By first determining whether lst is empty, the function returns the element of lst with the smallest value of func. The foldl function is applied to the remaining elements of the list, starting with the first one, if lst is not empty. The foldl function accepts a function as an input that compares two elements by applying the function func to them and returning the one with the smaller result of func. The function's output is the resultant element.

## Function 7

*;; sn-unfrndlst-user : (dict-of symbol (set-of symbol)) -> (cons symbol integer)*

*;;This function takes a graph as input, and returns the user with the least friends.*

(define (sn-unfrndlst-user graph)

  (let\* ([counts (map (lambda (user)

                        (cons user (set-count (dict-ref graph user empty))))

                      (dict-keys graph))])

    (let ([min-count-pair (min-by (lambda (x) (cdr x)) counts)])

      min-count-pair)))

I have defined the function sn-unfrndlst-user in the code above, and it only accepts one parameter:

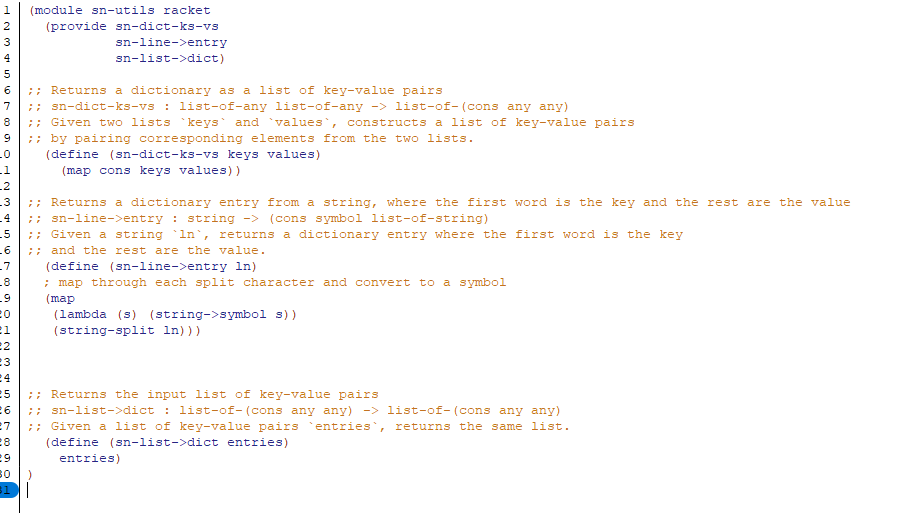
graph: a dictionary where the values stand in for the users' friends and the keys for the users themselves.

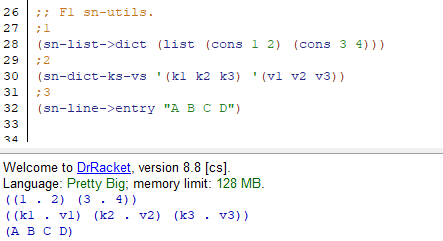
The function initially creates a list of pairs consisting of each user and the number of their friends by using the map function to map over the keys of the input graph. This produces the user with the fewest friends as the result. The function then uses the min-by function to determine the minimum count and returns the pair for that count. The min-by function requires two inputs: the input list itself and a function that extracts the value to be compared from each element of the input list. In this instance, a lambda function is utilised to retrieve the count value from each pair, and the list of pairs generated earlier.

# Results – Output Obtained

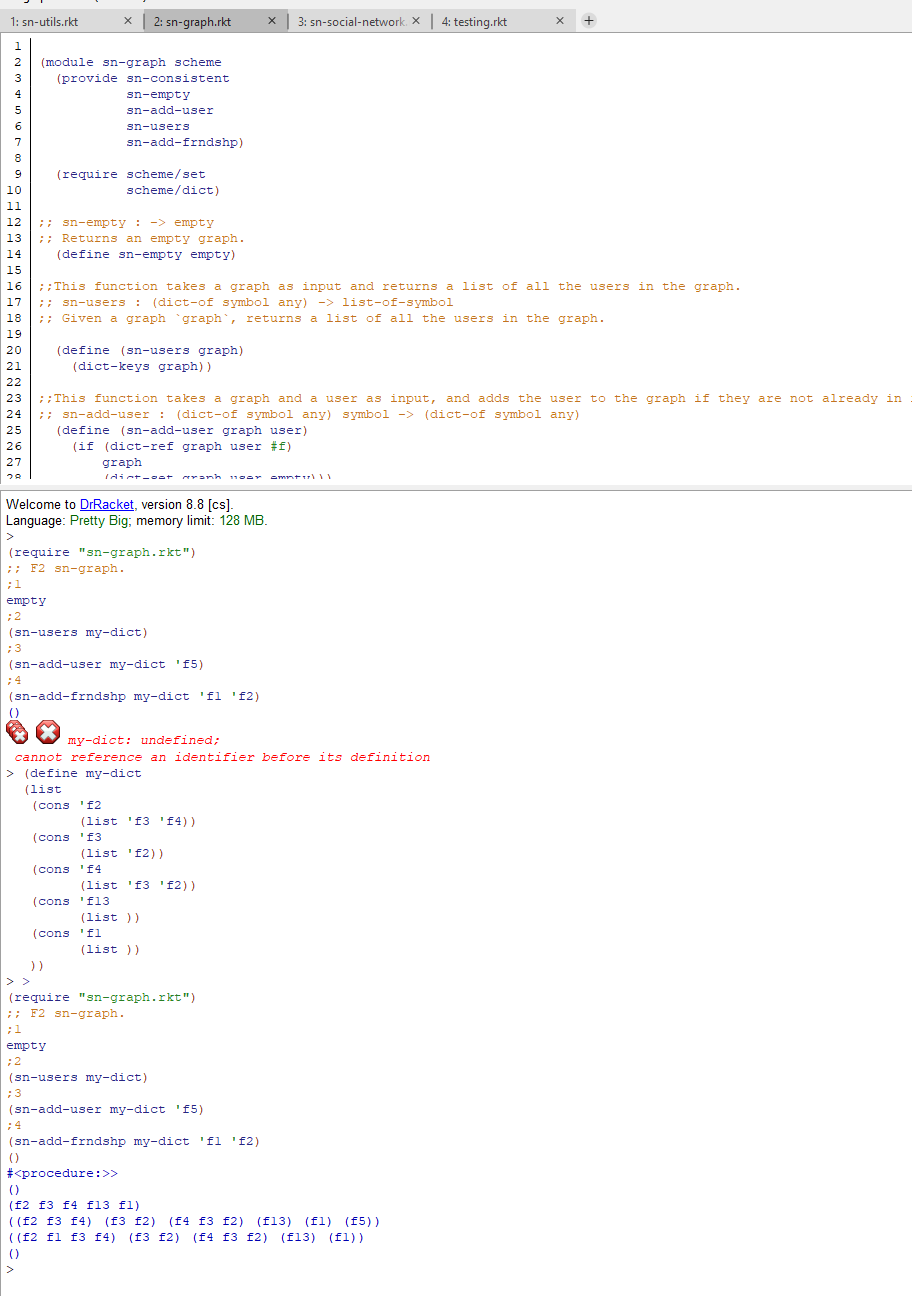
*Provide screenshots that demonstrate the results generated by running your code. That is show the output obtained in the REPL when calling your functions. Alternatively, you may simply cut and paste from the REPL.*

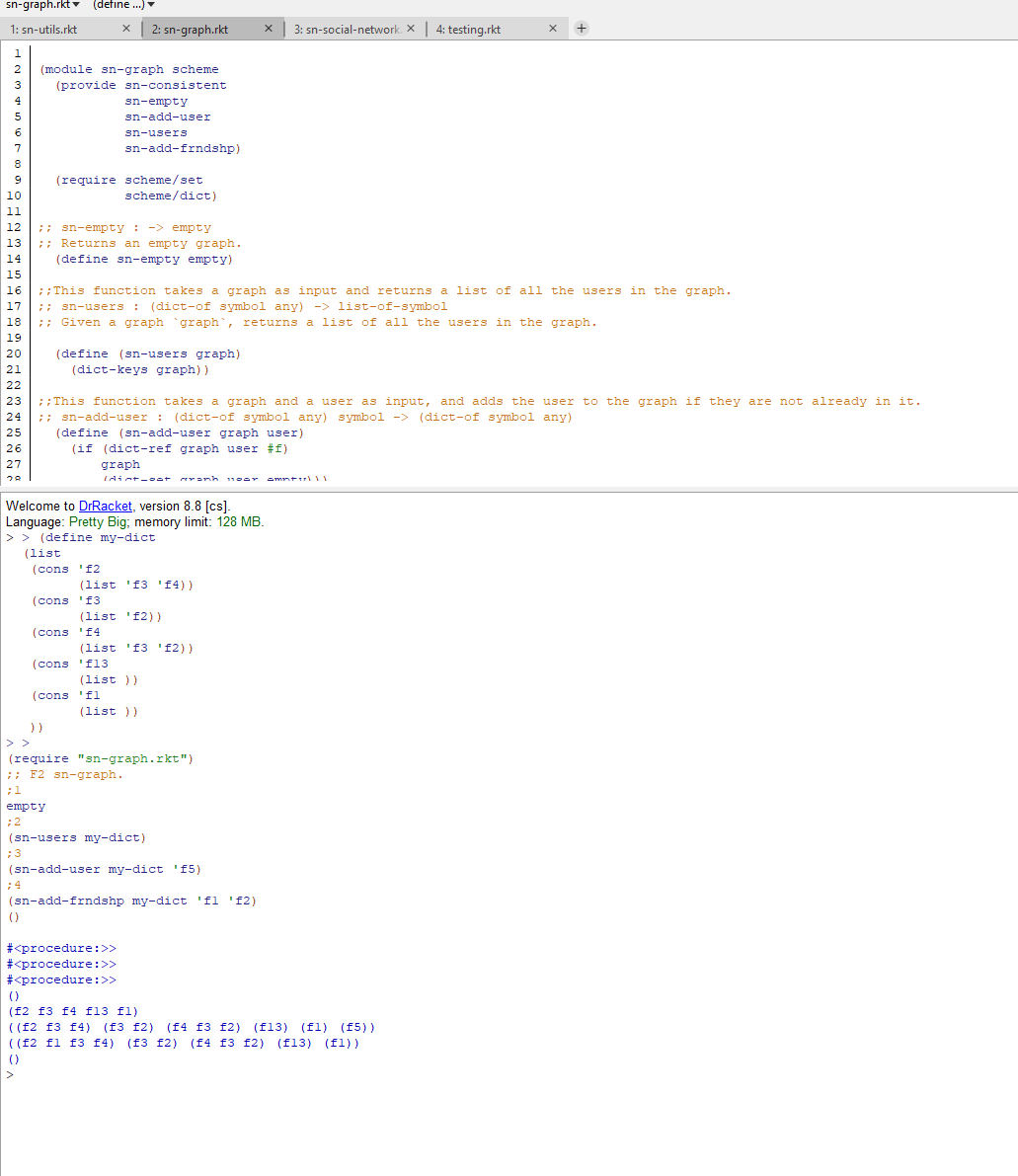
## TASK-1 util

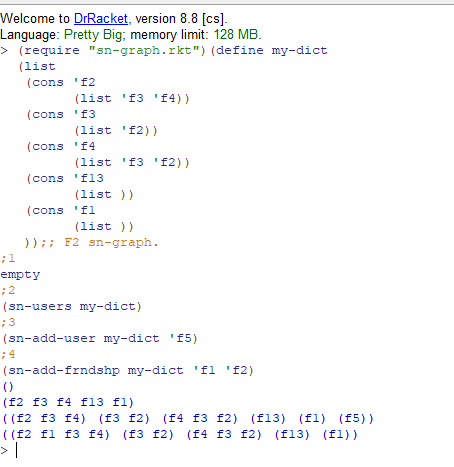




## TASK-2 grahp

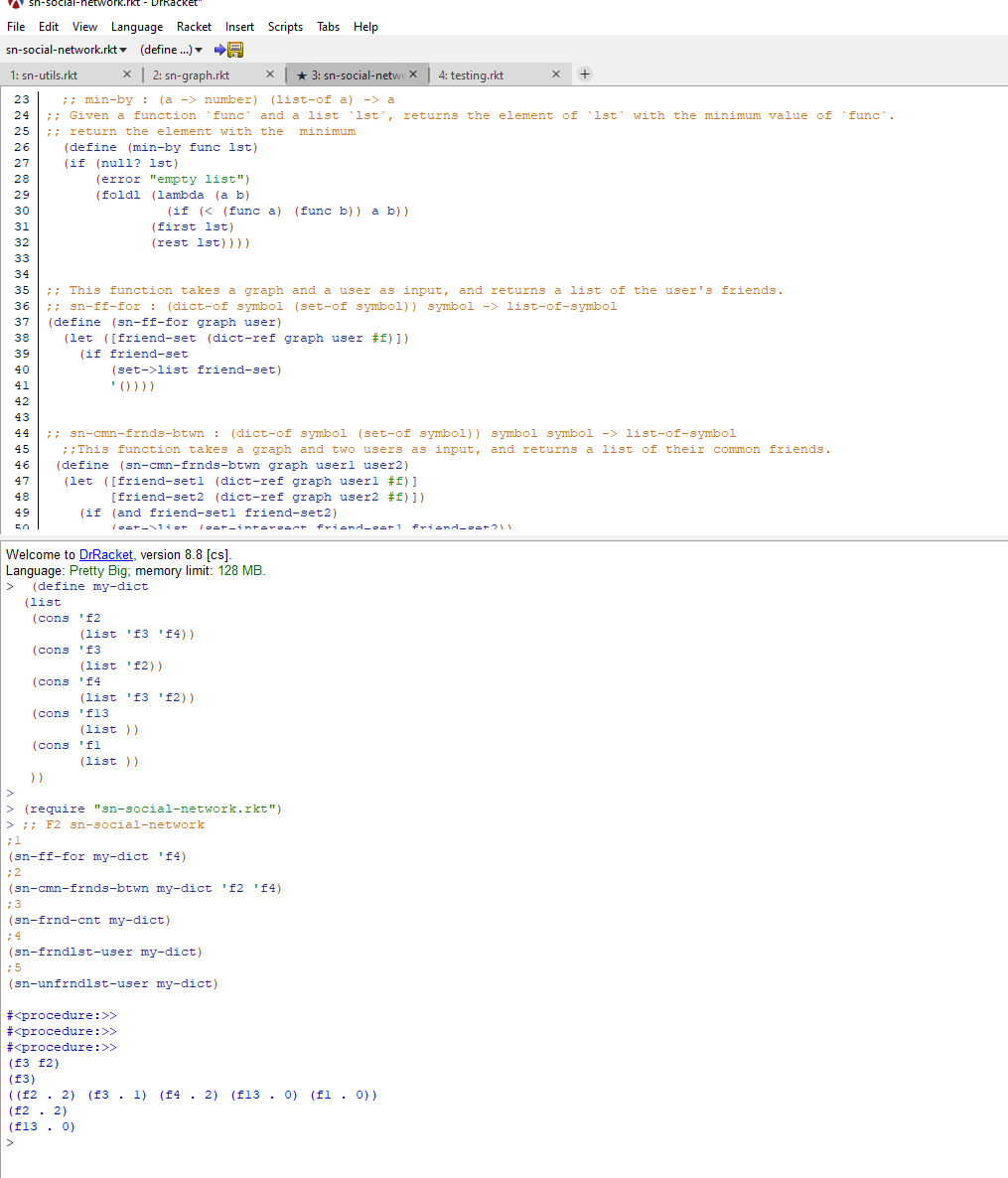


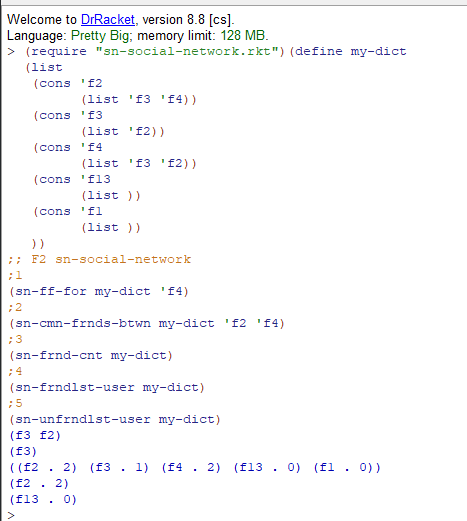




## TASK-SOCIAL NETWORKING







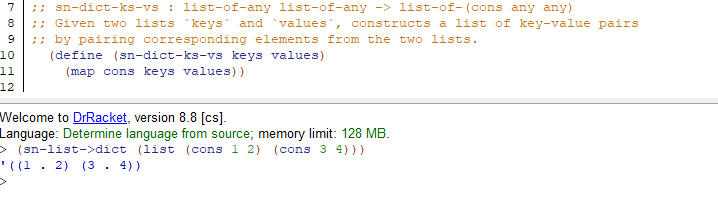
# TESTING

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

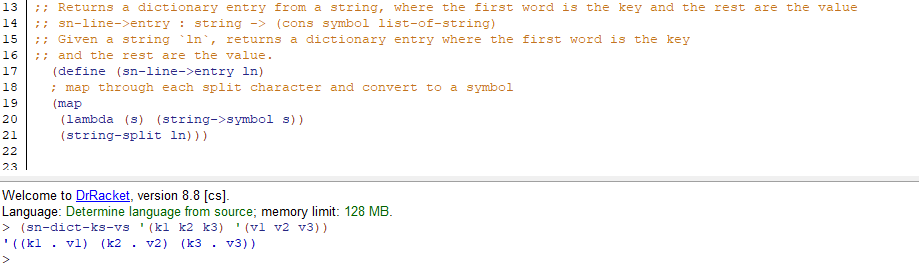
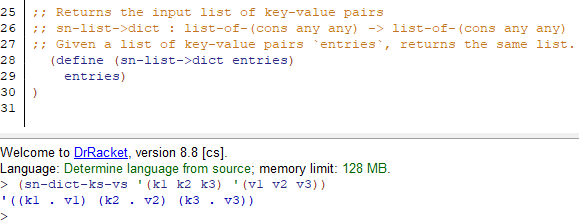
For this part I need to show my testing plan which I will show in two part one of my function fixing the code and then how I implement on the file app.

**File 1 Util**

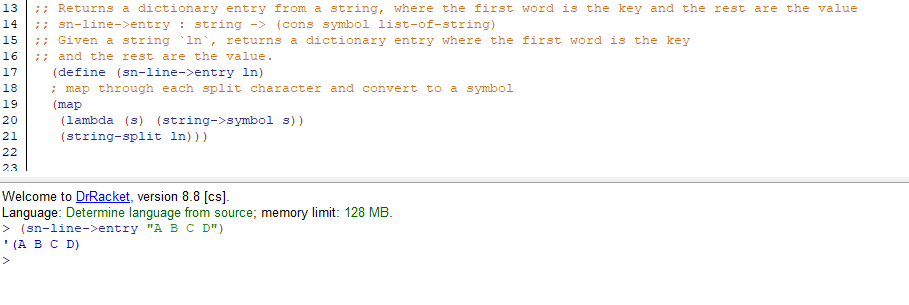
|  |  |
| --- | --- |
| **TestID** | T1 sn-dict-ks-vs |
| **Basic requirements** | Network need to be loaded |
| **Steps** | (sn-list->dict (list (cons 1 2) (cons 3 4))) |
| **Expected results** | If there are any mutual friends between users 1 and 2, it should provide a list of them. |
| **My results** | When there is at least one friend shared by both users, a non-empty list is returned. |



|  |  |
| --- | --- |
| **TestID** | T2 sn-dict-ks-vs |
| **Basic requirements** | Network need to be loaded |
| **Steps** | (sn-dict-ks-vs '(k1 k2 k3) '(v1 v2 v3)) |
| **Expected results** | Returns the input list of key-value pairs |
| **My results** | Given a list of key-value pairs `entries`, returns the same list |

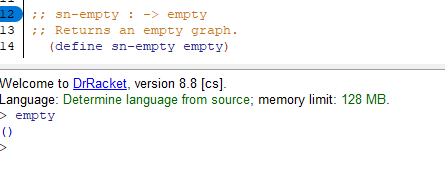


|  |  |
| --- | --- |
| **TestID** | T3 (sn-line->entry ) |
| **Basic requirements** | Network need to be loaded |
| **Steps** | (sn-line->entry "A B C D") |
| **Expected results** | Returns a dictionary entry from a string, where the first word is the key and the rest are the value |
| **My results** | Given a string `ln`, returns a dictionary entry where the first word is the key |

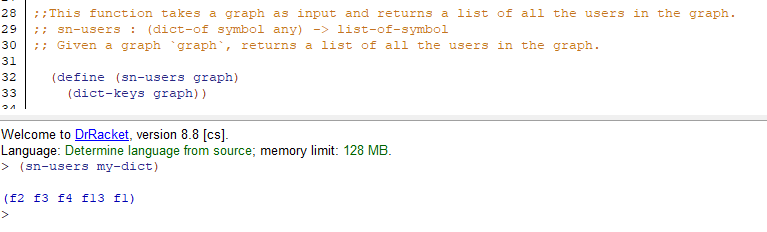


**File 2 Graphs**

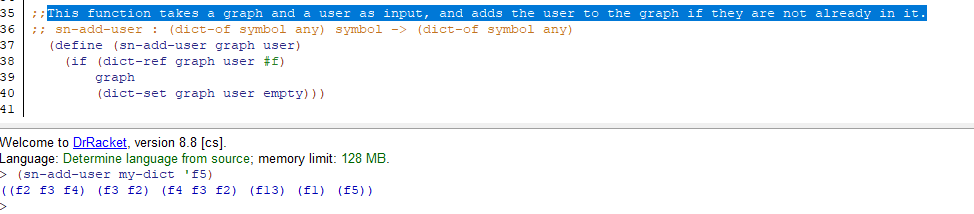
|  |  |
| --- | --- |
| **TestID** | T4 sn-empty |
| **Basic requirements** | Network need to be loaded |
| **Steps** | empty |
| **Expected results** | Returns an empty graph |
| **My results** | Returns an empty graph |

****

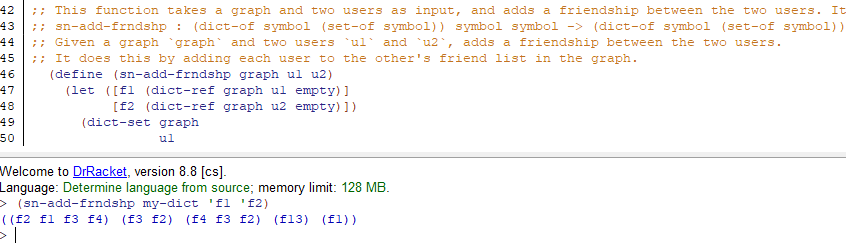
|  |  |
| --- | --- |
| **TestID** | T5 sn-users graph |
| **Basic requirements** | Network need to be loaded |
| **Steps** | (sn-users my-dict) |
| **Expected results** | Given a graph `graph`, returns a list of all the users in the graph |
| **My results** | Given a graph `graph`, returns a list of all the users in the graph |

****

|  |  |
| --- | --- |
| **TestID** | T6 sn-add-user |
| **Basic requirements** | Network need to be loaded |
| **Steps** | (sn-add-frndshp my-dict 'f1 'f2) |
| **Expected results** | takes a graph and two users as input, and adds a friendship between the two users. It does this by adding each user to the other's friend list in the graph |
| **My results** | Given a graph `graph` and two users `u1` and `u2`, adds a friendship between the two users |

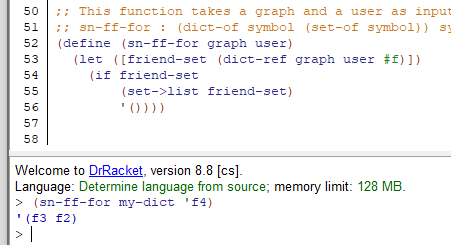
****

|  |  |
| --- | --- |
| **TestID** | T7 sn-add-frndshp |
| **Basic requirements** | Network need to be loaded |
| **Steps** | (sn-add-user my-dict 'f5) |
| **Expected results** | takes a graph and a user as input, and adds the user to the graph if they are not already in it. |
| **My results** | This function takes a graph and a user as input, and adds the user to the graph if they are not already in it. |

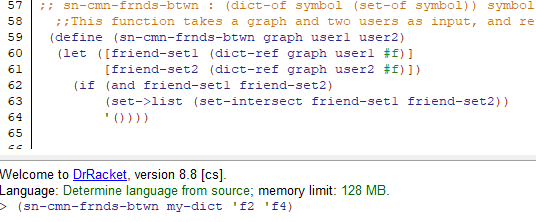
****

**File 3 network**

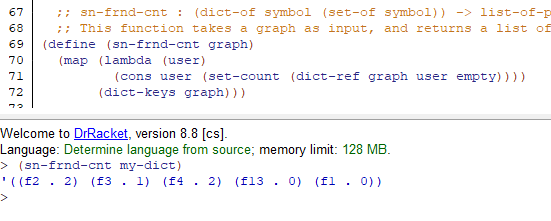
|  |  |
| --- | --- |
| **TestID** | T8 sn-ff-for |
| **Basic requirements** | Network need to be loaded |
| **Steps** | (sn-ff-for my-dict 'f4) |
| **Expected results** | a graph and a user as input, and returns a list of the user's friends |
| **My results** | This function takes a graph and a user as input, and returns a list of the user's friends |

****

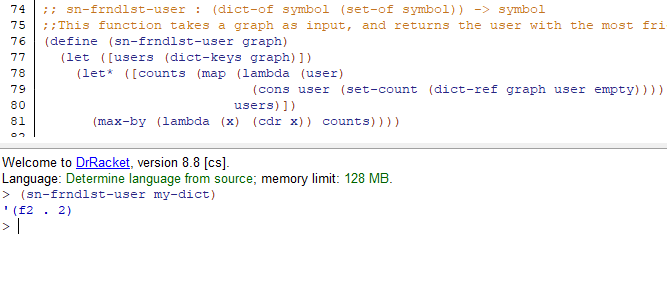
|  |  |
| --- | --- |
| **TestID** | T9 sn-cmn-frnds-btwn |
| **Basic requirements** | Network need to be loaded |
| **Steps** | (sn-cmn-frnds-btwn my-dict 'f2 'f4) |
| **Expected results** | ;This function takes a graph and two users as input, and returns a list of their common friends. |
| **My results** | ;This function takes a graph and two users as input, and returns a list of their common friends. |

****

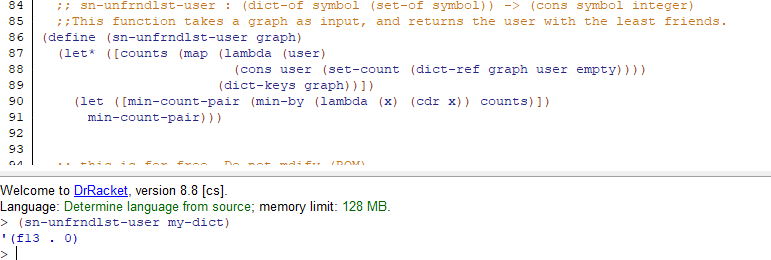
|  |  |
| --- | --- |
| **TestID** | T10 sn-frnd-cnt |
| **Basic requirements** | Network need to be loaded |
| **Steps** | (sn-frnd-cnt my-dict) |
| **Expected results** | This function takes a graph as input, and returns a list of users and their friend counts. |
| **My results** | This function takes a graph as input, and returns a list of users and their friend counts. |

****

|  |  |
| --- | --- |
| **TestID** | T11 sn-frndlst-user |
| **Basic requirements** | Network need to be loaded |
| **Steps** | (sn-frndlst-user my-dict) |
| **Expected results** | This function takes a graph as input, and returns the user with the most friends. |
| **My results** | This function takes a graph as input, and returns the user with the most friends. |

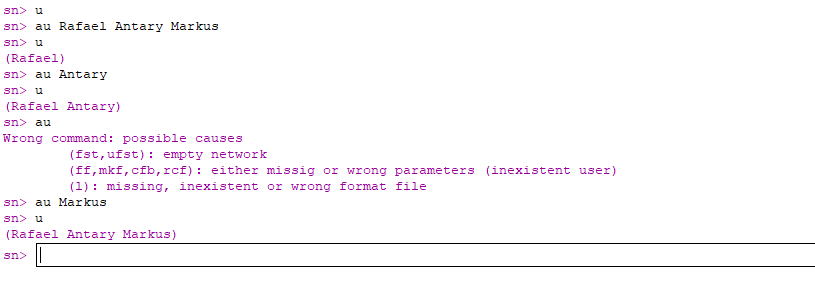
****

|  |  |
| --- | --- |
| **TestID** | T12 sn-unfrndlst-user |
| **Basic requirements** | Network need to be loaded |
| **Steps** | (sn-unfrndlst-user my-dict) |
| **Expected results** | This function takes a graph as input, and returns the user with the least friends. |
| **My results** | This function takes a graph as input, and returns the user with the least friends. |

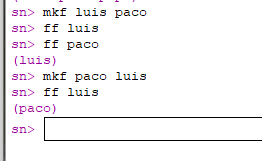
****

I require an end-to-end test to confirm that the procedures integrate as anticipated after the unit test on the broken procedures has been verified, as shown above. The network will be loaded from the file (sn2.txt) and subjected to a number of actions.

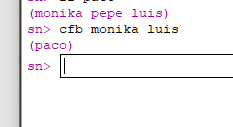
|  |  |
| --- | --- |
| **TestID** | T13 AddUser |
| **Basic requirements** | Network need to be loaded |
| **Steps** | Input **au [name/ID]** and press Enter |
| **Expected results** | If it didn't previously, the network should now have a new user entry. |
| **My results** | If a user with that name or ID does not already exist, a new entry is created; otherwise, the network remains as it is shown below. |



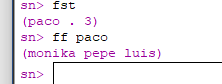
|  |  |
| --- | --- |
| **TestID** | T14 MakeFriends |
| **Basic requirements** | Network need to be loaded |
| **Steps** | Input **mkf [user1] [user2]** providing name/ID of existing network users  Press Enter |
| **Expected results** | The friends list of each user should be updated such that, if they aren't already, user1 will be on user2's friends list and vice versa. |
| **My results** | If the users were already friends, no modifications are done and both users' lists are updated accordingly. |



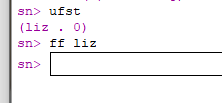
|  |  |
| --- | --- |
| **TestID** | T14 CommonFriendsBetween |
| **Basic requirements** | Network need to be loaded |
| **Steps** | Input **cfb [user1] [user2]** providing name/ID of existing network users  Press Enter |
| **Expected results** | If there are any mutual friends between users 1 and 2, it should provide a list of them. |
| **My results** | When there is at least one friend shared by both users, a non-empty list is returned. |



|  |  |
| --- | --- |
| **TestID** | T15 FriendliestUser |
| **Basic requirements** | Network need to be loaded |
| **Steps** | Input **fst** and press Enter |
| **Expected results** | Should return a pair that includes the name/Id and the number of friends of the user with the most friends. |
| **My results** | The name and number of friends of the user with the most friends are returned as a pair. Only one of them is given back if there are ties. |



|  |  |
| --- | --- |
| **TestID** | T16 UnfriendliestUser |
| **Basic requirements** | Network need to be loaded |
| **Steps** | Input **ufst** and press Enter |
| **Expected results** | Should return a pair that includes the name/Id and friend count of the user with the fewest friends. |
| **My results** | The name and number of friends of the user with the fewest friends are returned as a pair. Only one of them is given back if there are ties. |



# 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 did you learn* *from your experience? (not just about Scheme, but about programming in general, project development and time management, etc.)*
* *how would a similar task be completed differently?*
* *what can you carry forward to future development projects?*

Coming from an imperative background, thinking in functional programming was initially a little confusing for me, but I can now say that I am starting to get the feel of it. Through this coursework, I had the opportunity to engage with functional programming to address an issue that was almost exactly like one that would occur in practice. At first, it was challenging to consider fixes for the broken functions without considering variables and loops. It was simpler to determine whether strategy would work in place of a loop or a variable once the presenters provided a brief refresher.

I had the opportunity to learn a lot about functional programming while developing this program. For instance, despite the fact that programming is a relatively feared industry, I had no idea how profitable the expertise is. Also, I learned how crucial functional programming is in maintaining secure and reliable systems like WhatsApp, which was created using Erlang. For development project, I learned the value of planning, which entails outlining a strategy and assessing it before moving forward with actual development. I also learned how to save time by tackling almost trivial tasks first and then moving on to the more complex ones, though doing so naturally has a cost.

This method of development, which involves gradually working on little portions of the system at a time, seems appropriate for any software development endeavor. I'll have tactic as a talent for upcoming projects! Before beginning to code, it is crucial to have a plan for how to solve an issue. I learned to value the significance of starting small and increasing as I go. For any developer who wants to avoid breaking their monitor in the future, this is a crucial skill.