Project X

By Amin M00847012, Andrei M00808734 and Andres Leon

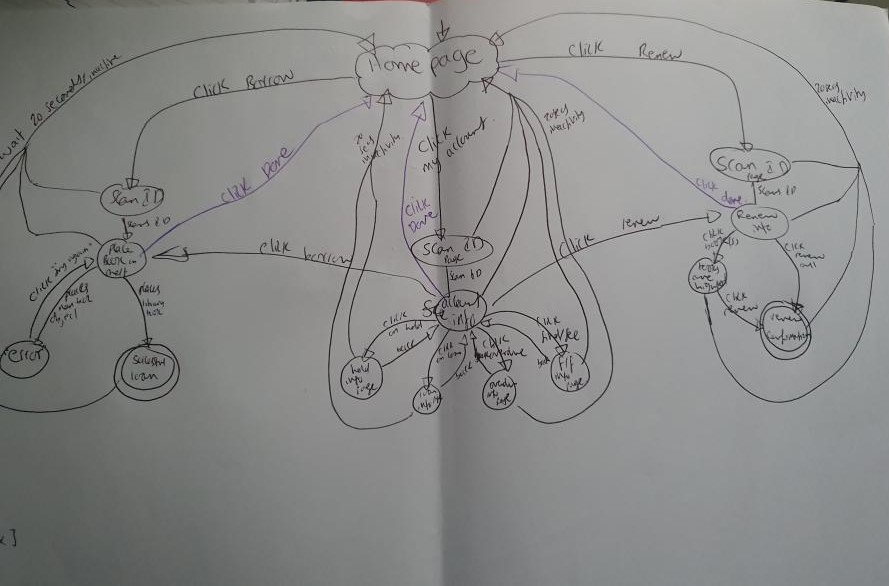
The FSM we picked

We decided to use the library machine as the base of our project . The machine consisted of multiple functions including borrowing , renewing , checking account information , placing hold orders and paying for overdue fees . This proved to be a challenge as there were many states and events to consider and implement in our FSM.

Design process

While drawing our FSM , we had to decide on difficult design choices based on convenience and simplicity. For example , we decided to have three separate ‘’Scan\_ID’’ states , as we thought of them as different pathways instead of leaving it as one state. We did not add the timer of 20 seconds that returns to the homepage in cases of inactivity , which is included in the real library machine. We assumed that the user knows exactly how to use the machine and why they are using the machine , meaning they won’t make any mistakes when operating the machine. However we did include a ‘’Done’’ button , which is also included in the real library machine in cases of human error or when they are finished with the machine. These are all design choices we have thought about in order replicate the library machine as close as possible but our goal wasn’t to exactly reproduce the actual machine so we didn’t include some functions as mentioned above.

A drawing of a tree

Description automatically generated with low confidence

FSM 2

FSM 1

Our code

(define fsm '(

(("a" "Click\_Borrow")"b")

This is a simple defined list in which we paired our states with the events and put output state all in one element , for the whole FSM. We decided to define our states as strings for simplicity’s sake .

This FSM list is used when we coded the next state in order for us to know what the next state will be after an event occurs at state x .

(("a" "Click\_My\_Account")"c")

(("a" "Click\_Renew")"d")

(("b" "ID\_Scanned")"e")

(("c" "ID\_Scanned")"f")

(("d" "ID\_Scanned")"g")

(("e" "Available\_Book")"h")

(("e" "Invalid\_Object")"i")

(("e" "Click Done")"a")

(("i" "Try\_Again")"e")

(("f" "Click")"j")

(("j" "Back") "f")

(("f" "Click")"k")

(("k" "Back")"f")

(("f" "Click")"l")

(("l" "Back")"f")

(("f" "Click")"m")

(("m" "Back")"f")

(("f" "Click\_Borrow")"e")

(("f" "Click\_Renew")"g")

(("f" "Click\_Done")"a")

(("g" "Click\_Books")"n")

(("g" "Click\_Renewall")"o")

(("g" "Click\_Done")"a")

(("n" "Click\_Renew")"o")))

(define conversion '(

("a" "Homepage")

This is a simple list of paired states with their more specific name so we can know what state relates to which string.

("b" "scan\_id")

("c" "scan\_id2")

("d" "scan\_id3")

("e" "Place\_Books\_On\_Shelf")

("f" "Account\_Info")

("g" "Renew\_Info")

("h" "Successfully\_Borrowed")

("i" "Error")

("j" "Hold")

("k" "On\_loan")

("l" "Items\_Overdue")

("m" "Fined\_Item")

("n" "Books\_Are\_Highlighted")

("o" "Renew\_Confirmation")))

This code takes 2 arguments. T is the state ( string ) we want to translate and P is the list which we will use to find the paired name of the state. It works by first checking if P is empty , if so it returns #F. Then checks if caar P (simplified state e.g. “a”) is equal to T , if so it returns cdar P ( translated state e.g. “Homepage”). This function is recursive as we added the function itself at the end , which will go through the whole list to try find if T and caar P are equal.

define translate (λ (t p)

(cond

((empty? p) #f)

((equal? (first(first p)) t ) (second (first p)))

(#t (translate t (rest p))))))

(define next-state (λ ( x y z)

(cond

((empty? z) #f)

((and (equal? (first(first(first z))) x) (equal? (second(first(first z))) y))(second(first z)))

(#t (next-state x y (rest z)))

(else "check inputs"))))

The next state function is used to find the next state after an event has occurred from a specified starting state. It takes three arguments x is the starting state , y is the event that occurs and z is the FSM list. It first checks if the FSM list is empty , if so it returns #F . It then checks if caaar z (simplified state )is equal to x and if cdaar z (event that occurs ) is equal to y . If both are true and it returns cdar z ( output state ). We added a recursive function by implementing next state and we used x y (rest z ) as the inputs , which would allow the function to search through the rest of list z and carry out the previous conditions.

Difficulties in the project

(define run-sequence (λ (init-state event-seq list)

(cond

((and (empty? event-seq) (empty? list)) "input valid event-seq and list")

((empty? event-seq) "input valid event-seq")

((empty? list) "input valid list")

((and (string? (first event-seq)) (list? list))(next-state init-state (first event-seq) list)

((run-sequence (next-state init-state (first event-seq) list) (first(rest event-seq)) list))))))

Notes : We were unable to finish coding a functional definition for run-sequence . We knew that run-sequence is basically next-state being used recursively with the previous output state being the new initial state and using the next event in the event list. However we were unable to transfer this idea to racket code. We particularly found recursion to be the difficult part when defining run-sequence and we hope to improve on this in the near future.   
  
  
**Sobs we believe we deserve :**

Sob 8

Sob 10

Sob 21

Sob 22

Sob 26

Sob 101

Sob 103

Sob 110

Sob 111

Sob 114