COMP1811 – Scheme Project Report

Name	Aysegul Kayikci	SID	001138854
Partner's name	NOTE: I had a pair, but he wanted to leave last time. Because of that situation, I did my coursework by myself.	SID	

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…

Functions with their expected outputs were already given so our design was like as it specified in coursework. We choose recursion over higher order programming because of recursion's flexibility. Also one of the main reason why I choose recursion over higher order programming is that I could not control higher order functions properly.

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.

2.1 CARD? ...

```
(define (is-inside? val lst)
 (list? (member val lst)))


(define (card? card)
 (and
  (is-inside? (cdr card) symbols)
  (or (is-inside? (car card) face) (is-inside? (car card) nums))
  ))
```

SUITE AND NUMERAL ...

```
(define (suite x) (cdr x))
(define (numeral y) (car y))
```

2.2 FACE? ...

```
(define (face? card) (is-inside? (car card) face)); Checks if the given pair's first value inside face
```

2.3 VALUE ...

```
(define (value card) (if (is-inside? (car card) nums) (car card) 0.5)); IF card is face return 0.5 else first number of the card
```

2.4 CARD->STRING ...

```
(define (card->string card) (string-append
                    (if (face? card) (string (car card)) (number->string (car card)))
                    (string (cdr card))))
```

```
;Makes first value of the card string then makes second string and appends them together
```

2.5 DECK?

```
(define (deck? lst) (cond
                [(null? lst) #t])
                [(card? (car lst)) (deck? (cdr lst))]
                [else #f]
 )
;checks if the list's first value is valid card
; if it is checks the rest
;until list empty
```

2.6 VALUEOF

```
(define (valueOf lst)
  (if (null? lst) 0 (+ (value (car lst)) (valueOf (cdr lst)))))
;first adds the first value of the list then asks for rest of it
```

2.7 DO-SUITE

```
(define (do-suite symbol)
  (define (make-cons-list lst symbol final)
  (cond
    [(null? lst) final]
    [else (make-cons-list (cdr lst) symbol (append final (list (cons (car lst) symbol))))]
    ))
  (make-cons-list (append nums face) symbol null))
;make cons from first value of the list with the given symbol and append it to final
;until list is empty
```

;if list is empty return final

2.8 DECK

```
(define deck (append (do-suite #\H) (do-suite #\C) (do-suite #\S) (do-suite #\D)))
```

2.9 DECK->STRINGS

```
(define (deck->strings deck)
  (map card->string deck))
;we apply card->string to each element of the deck
```

3.0 PROBABILITY

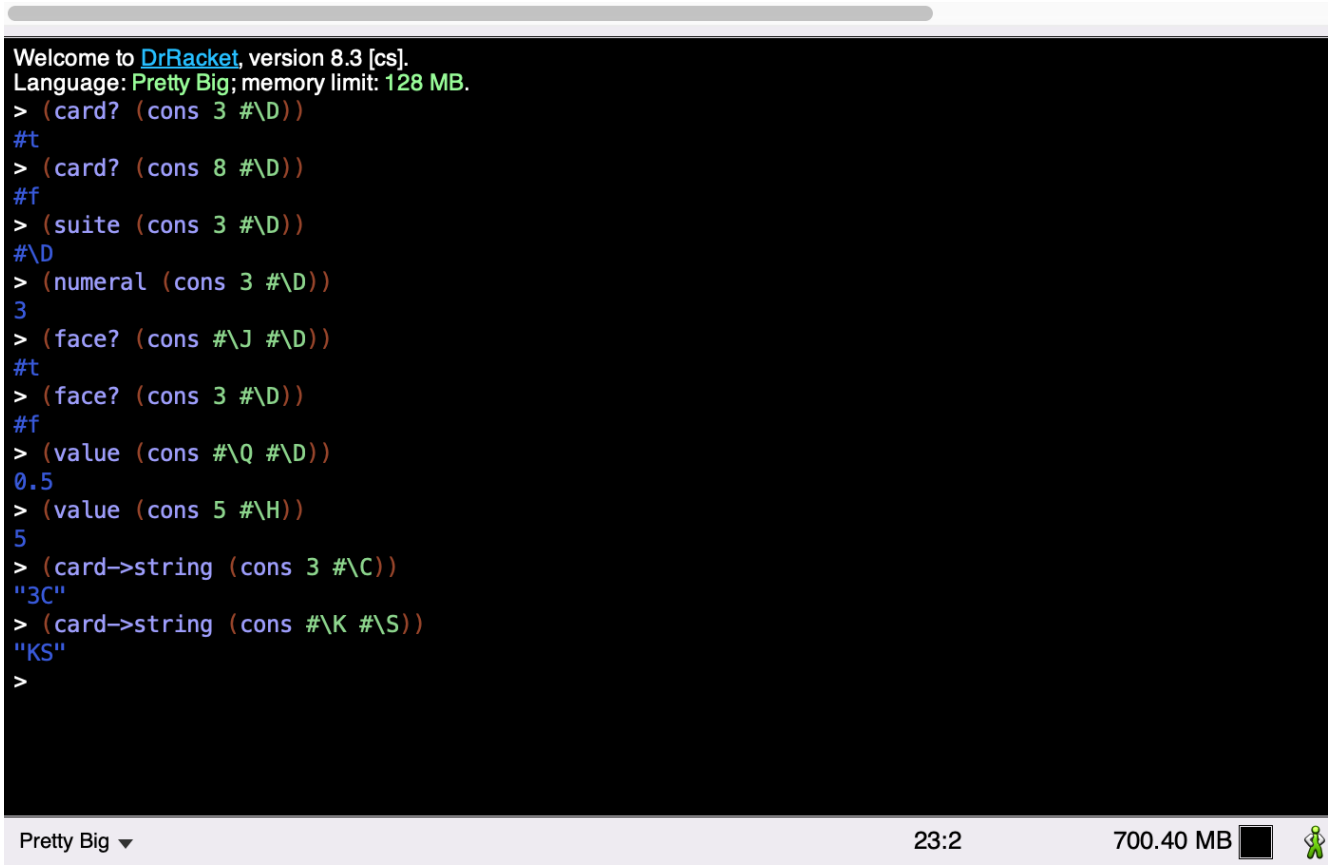
```
(define (probability comp num deck)
(define (get-probability comp num deck total)
  (cond
    [(null? deck) total]
    [(comp (value (car deck)) num) (get-probability comp num (cdr deck) (+ total 1))]
    [else (get-probability comp num (cdr deck) total)]))
  (get-probability comp num deck 0))

;We made internal function in order to make it tail recursion]
;Compare number with the first value of the given deck
;if comparison returns true increment total
;do this until list is empty
;if it is empty return total
```

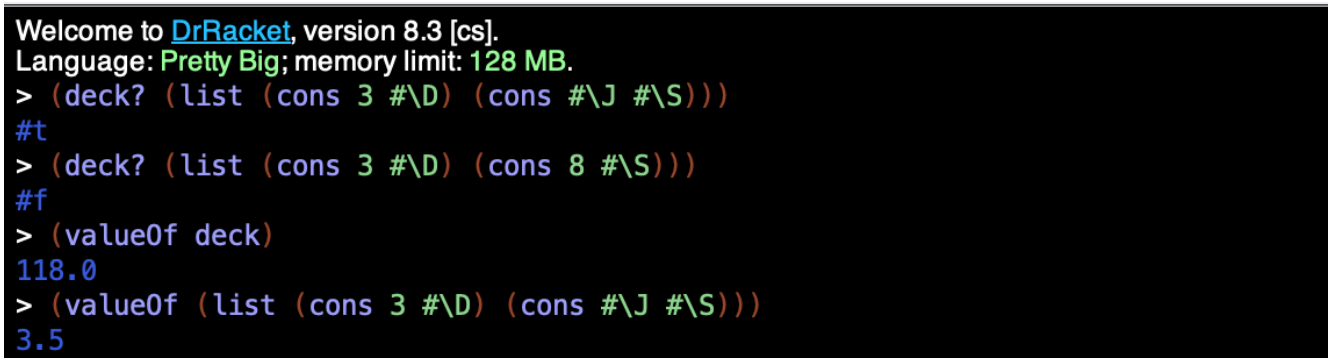
3. Results – Output Obtained

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.

3.1 TASK-1



3.2 TASK-2



```
> (do-suite #\D)
((1 . #\D)
 (2 . #\D)
 (3 . #\D)
 (4 . #\D)
 (5 . #\D)
 (6 . #\D)
 (7 . #\D)
 (#\J . #\D)
 (#\Q . #\D)
 (#\K . #\D))
> (do-suite #\J)
((1 . #\J)
 (2 . #\J)
 (3 . #\J)
 (4 . #\J)
 (5 . #\J)
 (6 . #\J)
 (7 . #\J)
 (#\J . #\J)
 (#\Q . #\J)
 (#\K . #\J))
```

```
> deck
((1 . #\H)
 (2 . #\H)
 (3 . #\H)
 (4 . #\H)
 (5 . #\H)
 (6 . #\H)
 (7 . #\H)
 (#\J . #\H)
 (#\Q . #\H)
 (#\K . #\H)
 (1 . #\C)
 (2 . #\C)
 (3 . #\C)
 (4 . #\C)
 (5 . #\C)
 (6 . #\C)
 (7 . #\C)
 (#\J . #\C)
 (#\Q . #\C)
 (#\K . #\C)
 (1 . #\S)
 (2 . #\S)
 (3 . #\S)
 (4 . #\S)
 (5 . #\S)
 (6 . #\S)
```

```
 (5 . #\S)
 (6 . #\S)
 (7 . #\S)
 (#\J . #\S)
 (#\Q . #\S)
 (#\K . #\S)
 (1 . #\D)
 (2 . #\D)
 (3 . #\D)
 (4 . #\D)
 (5 . #\D)
 (6 . #\D)
 (7 . #\D)
 (#\J . #\D)
 (#\Q . #\D)
 (#\K . #\D))
```

```
> (deck->strings deck)
("1H"
 "2H"
 "3H"
 "4H"
 "5H"
 "6H"
 "7H"
 "JH"
 "QH"
 "KH"
 "1C"
 "2C"
 "3C"
 "4C"
 "5C"
 "6C"
 "7C"
 "JC"
 "QC"
 "KC"
 "1S"
 "2S"
 "3S"
 "4S"
 "5S"
 "6S"
```

Pretty Big 38:10 398.79 MB

```
 "QC"
 "KC"
 "1S"
 "2S"
 "3S"
 "4S"
 "5S"
 "6S"
 "7S"
 "JS"
 "QS"
 "KS"
 "1D"
 "2D"
 "3D"
 "4D"
 "5D"
 "6D"
 "7D"
 "JD"
 "QD"
 "KD")
> (deck->strings (list (cons 3 #\D) (cons #\J #\H))
                 )
("3D" "JH")
>
```

4. TASK3

```
Welcome to DrRacket, version 8.3 [cs].
Language: Pretty Big; memory limit: 128 MB.
> (probability = 7 deck)
4
> (probability > 7 deck)
0
> (probability < 7 deck)
36
>
```

5. TESTING

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

My testing is done manually. I checked each functions works correctly as it is and I checked programme runs without a problem.

```
1  (define nums (list 1 2 3 4 5 6 7))
2  (define face (list #\J #\Q #\K))
3  (define symbols (list #\H #\C #\S #\D))
4  (require racket/list)
5
6  (define (is-inside? val lst)
7    (list? (member val lst)))
8  ;To check is the given value inside the given list
9
10 (define (card? card)
11   (and
12    (is-inside? (cdr card) symbols);First condition, cdr card is inside symbols
13    (or (is-inside? (car card) face) (is-inside? (car card) nums));To check car card inside nums or face
14    ))
15 ;To check given card is valid card
```

```
Welcome to DrRacket, version 8.3 [cs].
Language: Pretty Big; memory limit: 128 MB.
> (card? (cons 3 #\D))
#t
> (card? (cons 8 #\D))
#f
> (suite (cons 3 #\D))
#\D
> (numeral (cons 3 #\D))
3
> (face? (cons #\J #\D))
#t
> (face? (cons 3 #\D))
#f
> (value (cons #\Q #\D))
0.5
> (value (cons 5 #\H))
5
> (card->string (cons 3 #\C))
"3C"
> (card->string (cons #\K #\S))
"KS"
>
```

Pretty Big 23:2 700.40 MB

```
16
17 ;(define (name parameter) (body))
18 (define (suite x) (cdr x));returns the second value
19 (define (numeral y) (car y));returns the first value
20
21 (define (face? card) (is-inside? (car card) face)); Checks if the given pair's first value inside face
22 (define (value card) (if (is-inside? (car card) nums) (car card) 0.5)); IF card is face return 0.5 else first number of the card
23
24 (define (card->string card) (string-append
25                               (if (face? card) (string (car card)) (number->string (car card)))
26                               (string (cdr card))))
27 ;Makes first value of the card string then makes second string and appends them together
```

```
Welcome to DrRacket, version 8.3 [cs].
Language: Pretty Big; memory limit: 128 MB.
> (deck? (list (cons 3 #\D) (cons #\J #\S)))
#t
> (deck? (list (cons 3 #\D) (cons 8 #\S)))
#f
> (valueOf deck)
118.0
> (valueOf (list (cons 3 #\D) (cons #\J #\S)))
3.5
```

```
28
29 (define (deck? lst) (cond
30                       [(null? lst) #t])
31                       [(card? (car lst)) (deck? (cdr lst))]
32                       [else #f]
33   )
34 ;checks if the list's first value is valid card
35 ; if it is checks the rest
36 ;until list empty
37
38 (define (valueOf lst)
39   (if (null? lst) 0 (+ (value (car lst)) (valueOf (cdr lst)))))
40 ;first adds the first value of the list then asks for rest of it
41
42
```

```
> (do-suite #\D)
((1 . #\D)
 (2 . #\D)
 (3 . #\D)
 (4 . #\D)
 (5 . #\D)
 (6 . #\D)
 (7 . #\D)
 (#\J . #\D)
 (#\Q . #\D)
 (#\K . #\D))
> (do-suite #\J)
((1 . #\J)
 (2 . #\J)
 (3 . #\J)
 (4 . #\J)
 (5 . #\J)
 (6 . #\J)
 (7 . #\J)
 (#\J . #\J)
 (#\Q . #\J)
 (#\K . #\J))
```

```
> deck
((1 . #\H)
 (2 . #\H)
 (3 . #\H)
 (4 . #\H)
 (5 . #\H)
 (6 . #\H)
 (7 . #\H)
 (#\J . #\H)
 (#\Q . #\H)
 (#\K . #\H)
 (1 . #\C)
 (2 . #\C)
 (3 . #\C)
 (4 . #\C)
 (5 . #\C)
 (6 . #\C)
 (7 . #\C)
 (#\J . #\C)
 (#\Q . #\C)
 (#\K . #\C)
 (1 . #\S)
 (2 . #\S)
 (3 . #\S)
 (4 . #\S)
 (5 . #\S)
 (6 . #\S)
 (5 . #\S)
 (6 . #\S)
 (7 . #\S)
 (#\J . #\S)
 (#\Q . #\S)
 (#\K . #\S)
 (1 . #\D)
 (2 . #\D)
 (3 . #\D)
 (4 . #\D)
 (5 . #\D)
 (6 . #\D)
 (7 . #\D)
 (#\J . #\D)
 (#\Q . #\D)
 (#\K . #\D))
```

```
> (deck->strings deck)
("1H"
 "2H"
 "3H"
 "4H"
 "5H"
 "6H"
 "7H"
 "JH"
 "QH"
 "KH"
 "1C"
 "2C"
 "3C"
 "4C"
 "5C"
 "6C"
 "7C"
 "JC"
 "QC"
 "KC"
 "1S"
 "2S"
 "3S"
 "4S"
 "5S"
 "6S"
```

Pretty Big 38:10 398.79 MB

```
43
44 (define (do-suite symbol)
45   (define (make-cons-list lst symbol final)
46   (cond
47     [(null? lst) final]
48     [else (make-cons-list (cdr lst) symbol (append final (list (cons (car lst) symbol))))]
49     ))
50   (make-cons-list (append nums face) symbol null))
51 ;make cons from first value of the list with the given symbol and append it to final
52 ;until list is empty
53 ;if list is empty return final
54
```

```
 "QC"
 "KC"
 "1S"
 "2S"
 "3S"
 "4S"
 "5S"
 "6S"
 "7S"
 "JS"
 "QS"
 "KS"
 "1D"
 "2D"
 "3D"
 "4D"
 "5D"
 "6D"
 "7D"
 "JD"
 "QD"
 "KD")
> (deck->strings (list (cons 3 #\D) (cons #\J #\H))
                 )
("3D" "JH")
>
```

```
55 (define deck (append (do-suite #\H) (do-suite #\C) (do-suite #\S) (do-suite #\D)))
56
57 (define (deck->strings deck)
58
59 (define (make-string-deck deck total)
60   (cond
61     [(null? deck) total]
62     [else (make-string-deck (cdr deck) (append total (list (card->string (car deck)))))]))
63   (make-string-deck deck null))
64
65 (define (get-probability comp num deck total)
66   (if (null? deck)
67       total
68       (if
69        (comp (value (car deck)) num)
70        (get-probability comp num (cdr deck) (+ total 1))
71        (get-probability comp num (cdr deck) total)
72        )
73   )    )
74
75
76 (define (probability comp num deck) (get-probability comp num deck 0))
77
78 ;made internal function in order to make it tail recursion
79 ;compare number with the first value of the given
80 ;if comparison returns true invrement total
81 ;do this until list is empty
82 ;if it is empty return total
83
```

```
Welcome to DrRacket, version 8.3 [cs].
Language: Pretty Big; memory limit: 128 MB.
> (probability = 7 deck)
4
> (probability > 7 deck)
0
> (probability < 7 deck)
36
>
```

Play shuffle deck.

```
P(>7.5):0/40
P(<7.5):40/40
P(=7.5):0/40
HAND:()
VALUE:0
DECK:****...
accept
P(>7.5):27/39
P(<7.5):0/39
P(=7.5):12/39
HAND:(7D)
VALUE:7
DECK:****...
accept
P(>7.5):38/38
P(<7.5):0/38
P(=7.5):0/38
HAND:(QS 7D)
VALUE:7.5
DECK:****...
WIN
>
```

Define Cheat #t

```
Welcome to DrRacket, version 8.3 [cs].
Language: Pretty Big; memory limit: 128 MB.
> (define cheat #t)
> (play (shuffle deck) '())
#<procedure:>>
P(>7.5):0/40
P(<7.5):40/40
P(=7.5):0/40
HAND:()
VALUE:0
DECK:(7D KH 2C 6D)...
accept
P(>7.5):27/39
P(<7.5):0/39
P(=7.5):12/39
HAND:(7D)
VALUE:7
DECK:(KH 2C 6D QC)...
accept
P(>7.5):38/38
P(<7.5):0/38
P(=7.5):0/38
HAND:(KH 7D)
VALUE:7.5
DECK:(2C 6D QC 5S)...
WIN
>
```

6. 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?*

I really had problems understanding syntax in beginning but as I get proficient with the language, I started doing exercises. In coursework specifications it writes that this project should take around 4 hours, yet it took me weeks to finish. I believe the reasons why it took so much I had not any experience with scheme and functional programming before. I learnt how to make things from starch and not to rely on built in functions of the language. However, I need to mention that I had some help from someone my class. Because I worked alone and, in some points, I have gotten stuck some of the tasks. Also, I quite liked the functional programming and how it makes things side-effect free. Learning curve was linear, so beginning was not very frustrating. One of the biggest problems I encountered with was lack of resources on the internet.

I could have made the functions with using higher order programming or I could have done them with tail recursion yet most of the time I chose non-tail recursive functions because they were easier to

come up with. Also, if I went back in the time I would have prepared more. For future I could have add GUI to this project and I could have made the functions much smaller and more efficient than they are currently. I could have added modes, but I did not because it was not specified in coursework specifications.

7. GROUP PRO FORMA

Describe the division of work and agree percentage contributions. The pro forma must be signed by all group members and an identical copy provided in each report. If you cannot agree percentage contributions, please indicate so in the notes column and provide your reasoning.
(THIS SECTION SHOULD BE THE SAME FOR BOTH PARTNERS)

Partner ID	Tasks/Features Completed	%Contribution	Signature	Notes
1	Aysegul Kayikci	%100	AK	**I had a pair, but he wanted to leave last time. Because of that situation, I did my coursework by myself.**
	Total	%100		