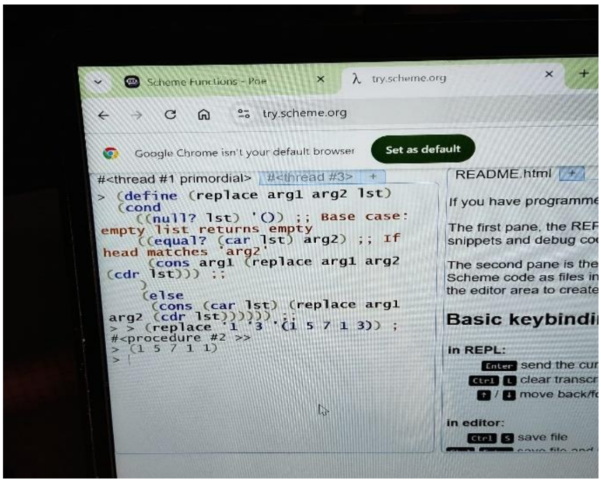
HW4 PROGLANG **Tatyana Selemenchuk 220504542**

(*All my handwritings you can find below)

1. Takes three arguments where the first element replaces of the occurrences of the second argument in the list which is the last argument

```
(define (replace arg1 arg2 lst)
  (cond
    ((null? lst) '()) ;; Base case: empty list returns empty
    ((equal? (car lst) arg2) ;; If head matches 'arg2'
     (cons arg1 (replace arg1 arg2 (cdr lst))) ;;
    )
    (else
     (cons (car lst) (replace arg1 arg2 (cdr lst)))))) ;;
> (replace 1 3 '(1 5 7 1 3)) ;
```



```
> (define (replace arg1 arg2 lst)
  (cond
    ((null? lst) '()) ;; Base case:
empty list returns empty
    ((equal? (car lst) arg2) ;; If
head matches 'arg2'
      (cons arg1 (replace arg1 arg2
(cdr lst))) ;;
     )
     (else
      (cons (car lst) (replace arg1
arg2 (cdr lst)))))) ;;
> > (replace '1 '3 '(1 5 7 1 3)) ;
#<procedure #2 >>
> (1 5 7 1 1)
> |
```

If you have programme

The first pane, the REP
snippets and debug co

The second pane is the
Scheme code as files in
the editor area to create

Basic keybindi

in REPL:

Enter send the cur
Ctrl L clear transcr
↑ / ↓ move back/f

in editor:

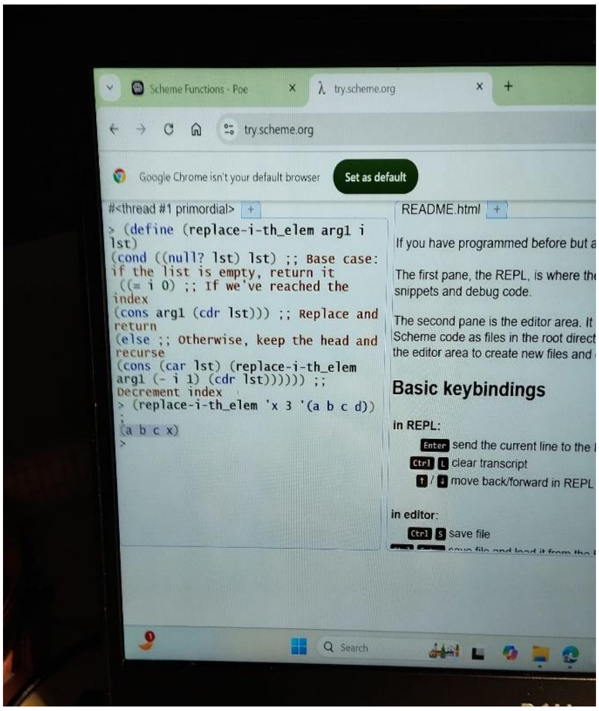
Ctrl S save file

> (1 5 7 1 1)

2. Takes three arguments where the first argument replaces the ith element (second argument) in the list (the third argument)

```
(define (replace-i-th_elem arg1 i lst)
(cond ((null? lst) lst) ;; Base case: if the list is empty, return it
 ((= i 0) ;; If we've reached the index
(cons arg1 (cdr lst))) ;; Replace and return
(else
```

(cons (car lst) (replace-i-th_elem arg1 (- i 1) (cdr lst)))))) ;; Decrement index

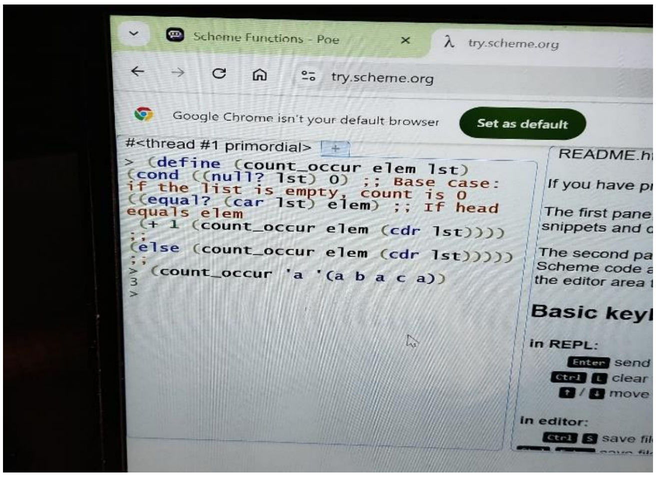


```
#<thread #1 primordial>
> (define (replace-i-th_elem arg1 i
lst)
(cond ((null? lst) lst) ;; Base case:
if the list is empty, return it
 ((= i 0) ;; If we've reached the
index
(cons arg1 (cdr lst))) ;; Replace and
return
(else ;; Otherwise, keep the head and
recurse
(cons (car lst) (replace-i-th_elem
arg1 (- i 1) (cdr lst)))))) ;;
Decrement index
> (replace-i-th_elem 'x 3 '(a b c d))
;
(a b c x)
>
```

(replace-i-th_elem 'x 3 '(a b c d)) **; > (a b c x)**

3. takes two arguments where the number of occurrences of the first argument in the list (second argument) is counted.

(define (count_occur elem lst)
(cond ((null? lst) 0) ;; Base case: if the list is empty, count is 0
((equal? (car lst) elem) ;; If head equals elem
 (+ 1 (count_occur elem (cdr lst))))
(else (count_occur elem (cdr lst)))))

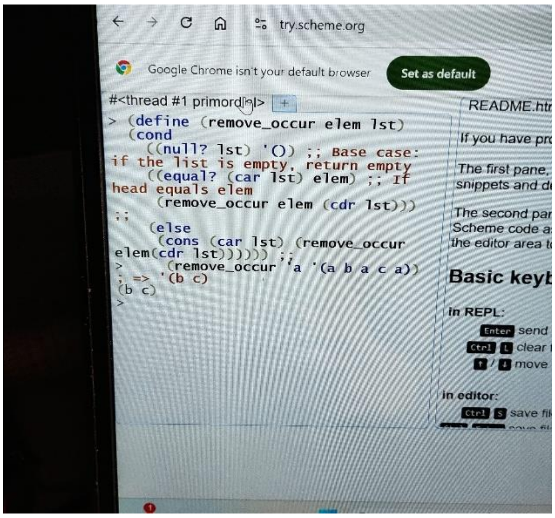


```
#<thread #1 primordial>
> (define (count_occur elem lst)
(cond ((null? lst) 0) ;; Base case:
if the list is empty, count is 0
((equal? (car lst) elem) ;; If head
equals elem
 (+ 1 (count_occur elem (cdr lst))))
;;
(else (count_occur elem (cdr lst)))))
;;
> (count_occur 'a '(a b a c a))
3
>
```

(count_occur 'a '(a b a c a**)) => 3**

4. takes two arguments where all the occurrences of the first argument in the list (second argument) is removed.

```
(define (remove_occur elem lst)
  (cond
    ((null? lst) '()) ;; Base case: if the list is empty, return empty
    ((equal? (car lst) elem) ;; If head equals elem
     (remove_occur elem (cdr lst))) ;;
    (else
     (cons (car lst) (remove_occur elem(cdr lst)))))) ;;
```

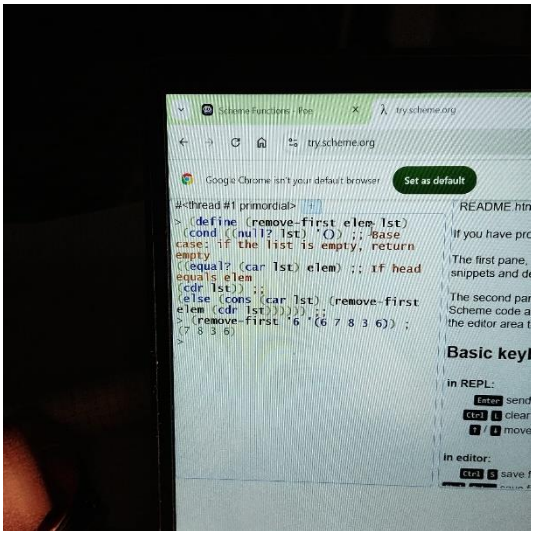


```
#<thread #1 primordial>
> (define (remove_occur elem lst)
  (cond
    ((null? lst) '()) ;; Base case:
if the list is empty, return empty
    ((equal? (car lst) elem) ;; If
head equals elem
     (remove_occur elem (cdr lst)))
;;
    (else
     (cons (car lst) (remove_occur
elem(cdr lst)))))) ;;
>     (remove_occur 'a '(a b a c a))
; => '(b c)
(b c)
>
```

(remove_occur 'a '(a b a c a)) ; > **'(b c)**

5. takes two arguments where only the first occurrence (the first argument) in the list(second argument) is removed.

(define (remove-first elem lst)
(cond ((null? lst) '()) ;; Base case: if the list is empty, return empty
((equal? (car lst) elem) ;; If head equals elem
(cdr lst)) ;;
(else (cons (car lst) (remove-first elem (cdr lst))))))

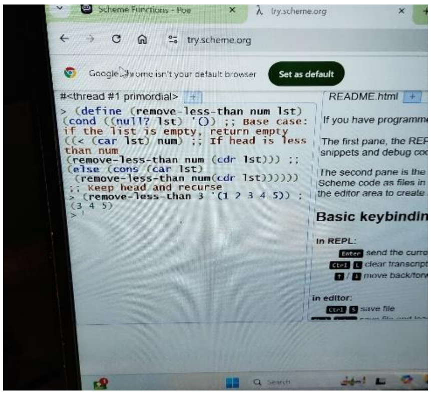


```
#<thread #1 primordial>
> (define (remove-first elem lst)
 (cond ((null? lst) '()) ;; Base
case: if the list is empty, return
empty
((equal? (car lst) elem) ;; If head
equals elem
(cdr lst)) ;;
(else (cons (car lst) (remove-first
elem (cdr lst)))))) ;;
> (remove-first '6 '(6 7 8 3 6)) ;
(7 8 3 6)
>
```

(remove-first '6 '(6 7 8 3 6)) **=> (7 8 3 6)**

6. takes two arguments where all the occurrences that are less than the first argument in the list (second argument) is removed.

(define (remove-less-than num lst)
(cond ((null? lst) '()) ;; Base case: if the list is empty, return empty
((< (car lst) num) ;; If head is less than num
(remove-less-than num (cdr lst)))
(else (cons (car lst)
(remove-less-than num(cdr lst))))))

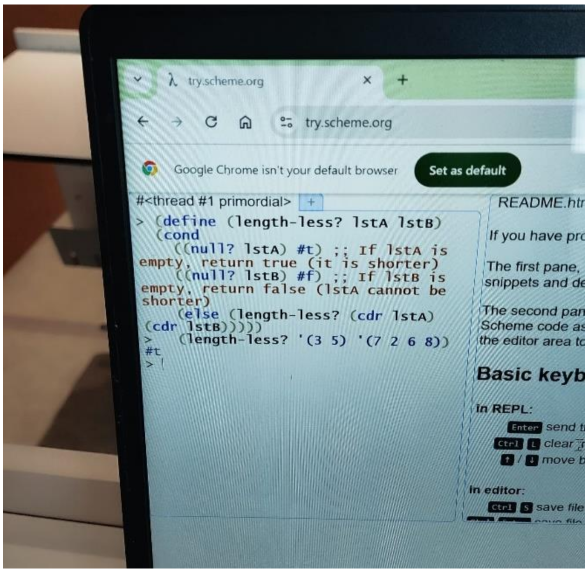


```
> (define (remove-less-than num lst)
(cond ((null? lst) '()) ;; Base case:
if the list is empty, return empty
((< (car lst) num) ;; If head is less
than num
(remove-less-than num (cdr lst))) ;;
(else (cons (car lst)
 (remove-less-than num(cdr lst))))))
;; Keep head and recurse
> (remove-less-than 3 '(1 2 3 4 5)) ;
(3 4 5)
>
```

(remove-less-than 3 '(1 2 3 4 5)) **; > (3 4 5)**

7. takes two arguments and evaluates to true if the length of list A (first argument) is less than the length of list B (second argument).

```
> (define (length-less? lstA lstB)
  (cond
    ((null? lstA) #t) ;; If lstA is empty, return true (it is shorter)
    ((null? lstB) #f) ;; If lstB is empty, return false (lstA cannot be shorter)
    (else (length-less? (cdr lstA) (cdr lstB)))))
```



```
> (define (length-less? lstA lstB)
   (cond
     ((null? lstA) #t) ;; If lstA is
empty, return true (it is shorter)
     ((null? lstB) #f) ;; If lstB is
empty, return false (lstA cannot be
shorter)
     (else (length-less? (cdr lstA)
(cdr lstB)))))
>    (length-less? '(3 5) '(7 2 6 8))
#t
>
```

```
>   (length-less? '(3 5) '(7 2 6 8))
```

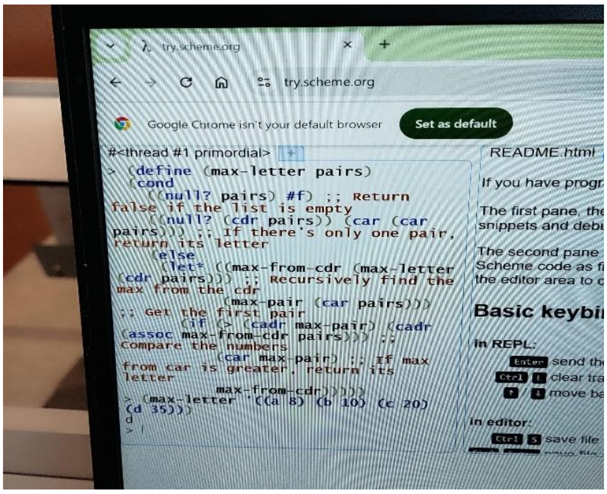
#t

```
>  (length-less? '(1 2 3) '(4 5))
```

 #f

8. takes a list of pairs where each pair is a letter and a positive number (associated with it) and evaluates to the letter whose associated number is the biggest.

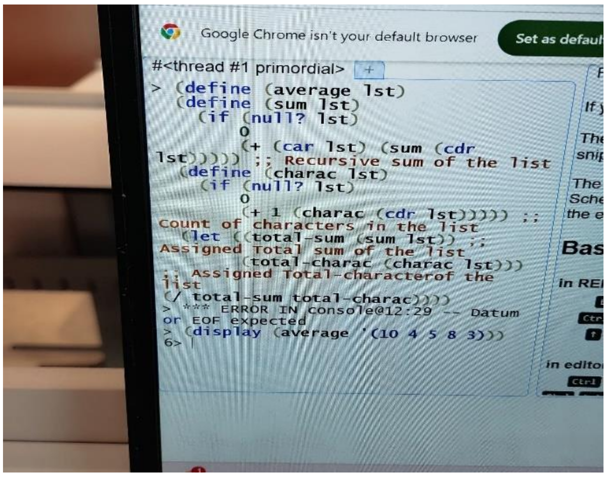
```
> (define (max-letter pairs)
   (cond
     ((null? pairs) #f) ;; Return false if the list is empty
     ((null? (cdr pairs)) (car (car pairs))) ;; If there's only one pair, return its letter
     (else
     (let* ((max-from-cdr (max-letter (cdr pairs))) ;; Recursively find the max from the cdr
            (max-pair (car pairs))) ;; Get the first pair
       (if (> (cadr max-pair) (cadr (assoc max-from-cdr pairs))) ;; Compare the numbers
        (car max-pair) ;; If max from car is greater, return its letter
        max-from-cdr)))))
> (max-letter '((a 8) (b 10) (c 20) (d 35))) => d
```



```
#<thread #1 primordial>
> (define (max-letter pairs)
  (cond
    ((null? pairs) #f) ;; Return
false if the list is empty
    ((null? (cdr pairs)) (car (car
pairs))) ;; If there's only one pair,
return its letter
    (else
      (let* ((max-from-cdr (max-letter
(cdr pairs))) ;; Recursively find the
max from the cdr
             (max-pair (car pairs)))
;; Get the first pair
        (if (> (cadr max-pair) (cadr
(assoc max-from-cdr pairs))) ;;
Compare the numbers
            (car max-pair) ;; If max
from car is greater, return its
letter
            max-from-cdr)))))
> (max-letter '((a 8) (b 10) (c 20)
(d 35)))
d
> |
```

9. takes a list of integers and evaluates to the average of the list.

```
> (define (average lst)
  (define (sum lst)
    (if (null? lst)
        0
        (+ (car lst) (sum (cdr lst))))) ;; Recursive sum of the list
  (define (charac lst)
    (if (null? lst)
        0
        (+ 1 (charac (cdr lst))))) ;; Count of characters in the list
  (let ((total-sum (sum lst)) ;; Assigned Total sum of the list
        (total-charac (charac lst))) ;; Assigned Total-characterof the list
(/ total-sum total-charac))))
```

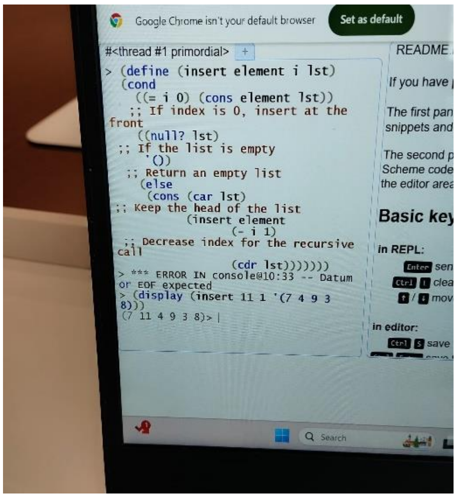


```
#<thread #1 primordial>
> (define (average lst)
  (define (sum lst)
    (if (null? lst)
        0
        (+ (car lst) (sum (cdr
lst))))) ;; Recursive sum of the list
  (define (charac lst)
    (if (null? lst)
        0
        (+ 1 (charac (cdr lst))))) ;;
Count of characters in the list
  (let ((total-sum (sum lst)) ;;
Assigned Total sum of the list
        (total-charac (charac lst)))
;; Assigned Total-characterof the
list
(/ total-sum total-charac))))
> *** ERROR IN console@12:29 -- Datum
or EOF expected
> (display (average '(10 4 5 8 3)))
6> |
```

```
>
(display (average '(10 4 5 8 3)))
6>
```

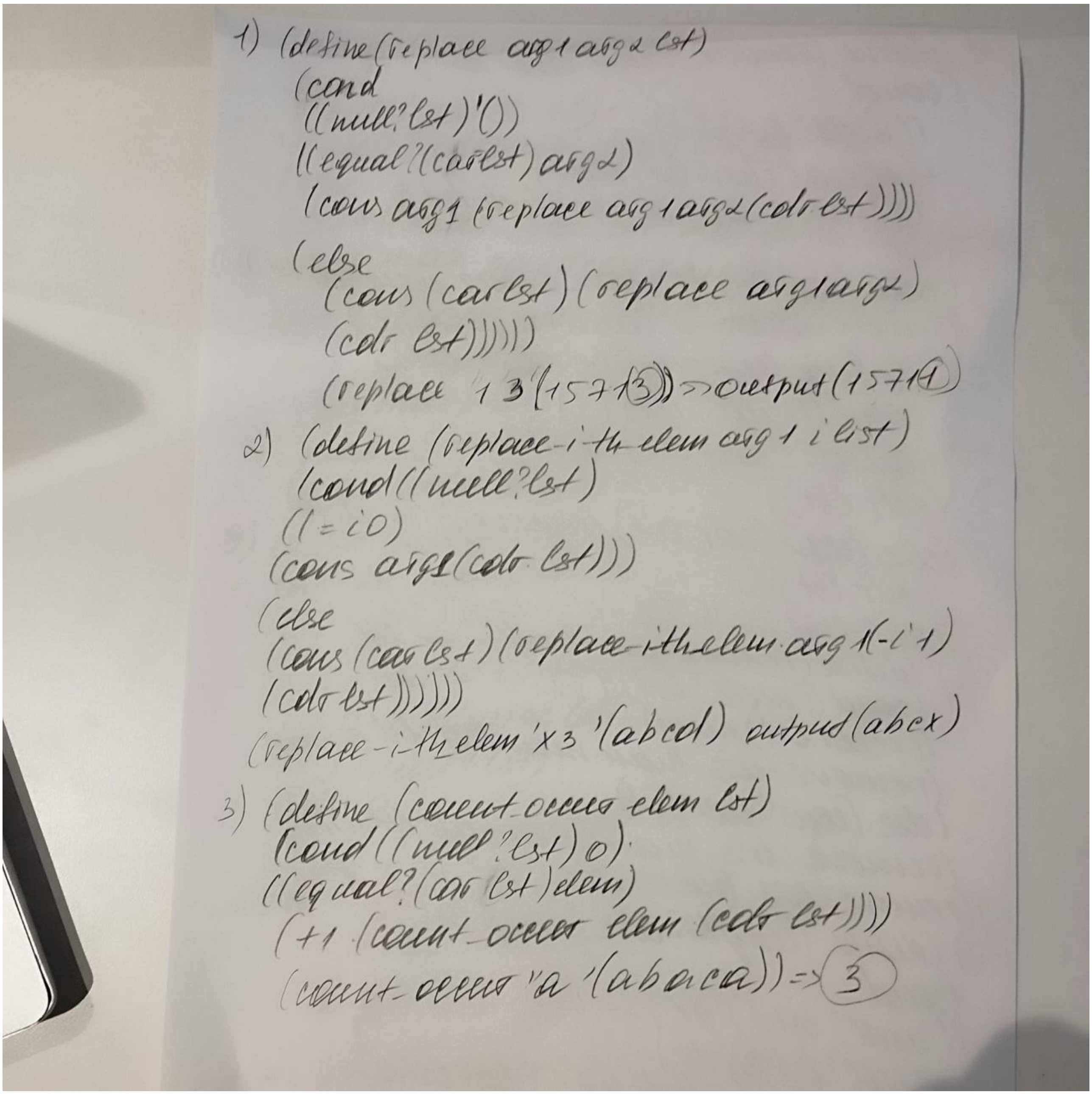
10. takes three arguments where the first argument inserted in the ith position (second argument) in the list (the third argument).

```
(define (insert element i lst)
 (cond
  ((= i 0) (cons element lst))      ;; If index is 0, insert at the front
  ((null? lst)              ;; If the list is empty
   '())                  ;; Return an empty list
  (else
   (cons (car lst)           ;; Keep the head of the list
       (insert element
           (- i 1)       ;; Decrease index for the recursive call
           (cdr lst))))))
> (display (insert 11 1 '(7 4 9 3 8)))
```



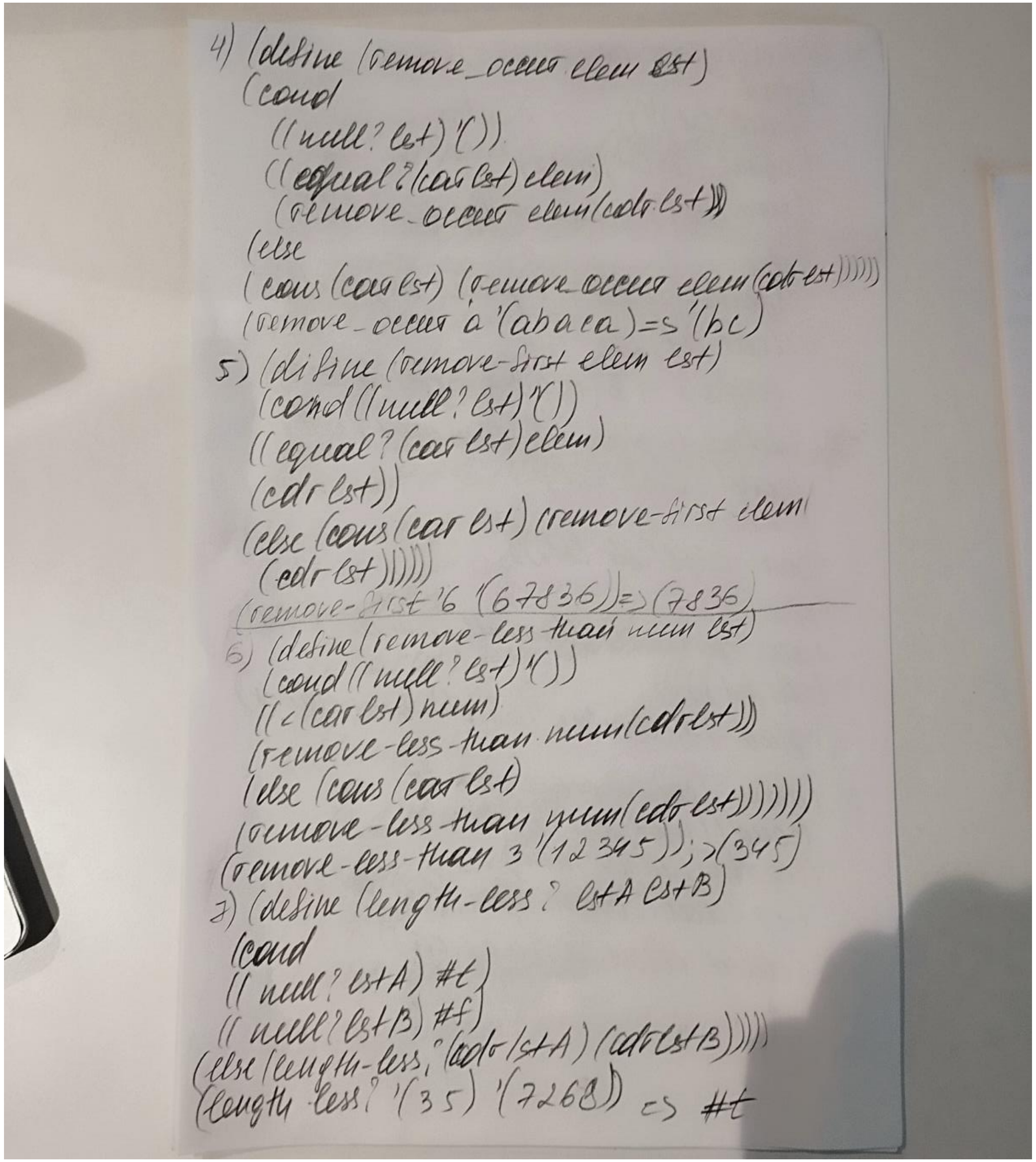
```
> (define (insert element i lst)
  (cond
    ((= i 0) (cons element lst))
   ;; If index is 0, insert at the
front
     ((null? lst)
 ;; If the list is empty
      '())
  ;; Return an empty list
     (else
      (cons (car lst)
;; Keep the head of the list
            (insert element
                    (- i 1)
 ;; Decrease index for the recursive
call
                    (cdr lst)))))))
> *** ERROR IN console@10:33 -- Datum
or EOF expected
> (display (insert 11 1 '(7 4 9 3
8)))
(7 11 4 9 3 8)> |
```

Out put (7 11 4 9 3 8)>

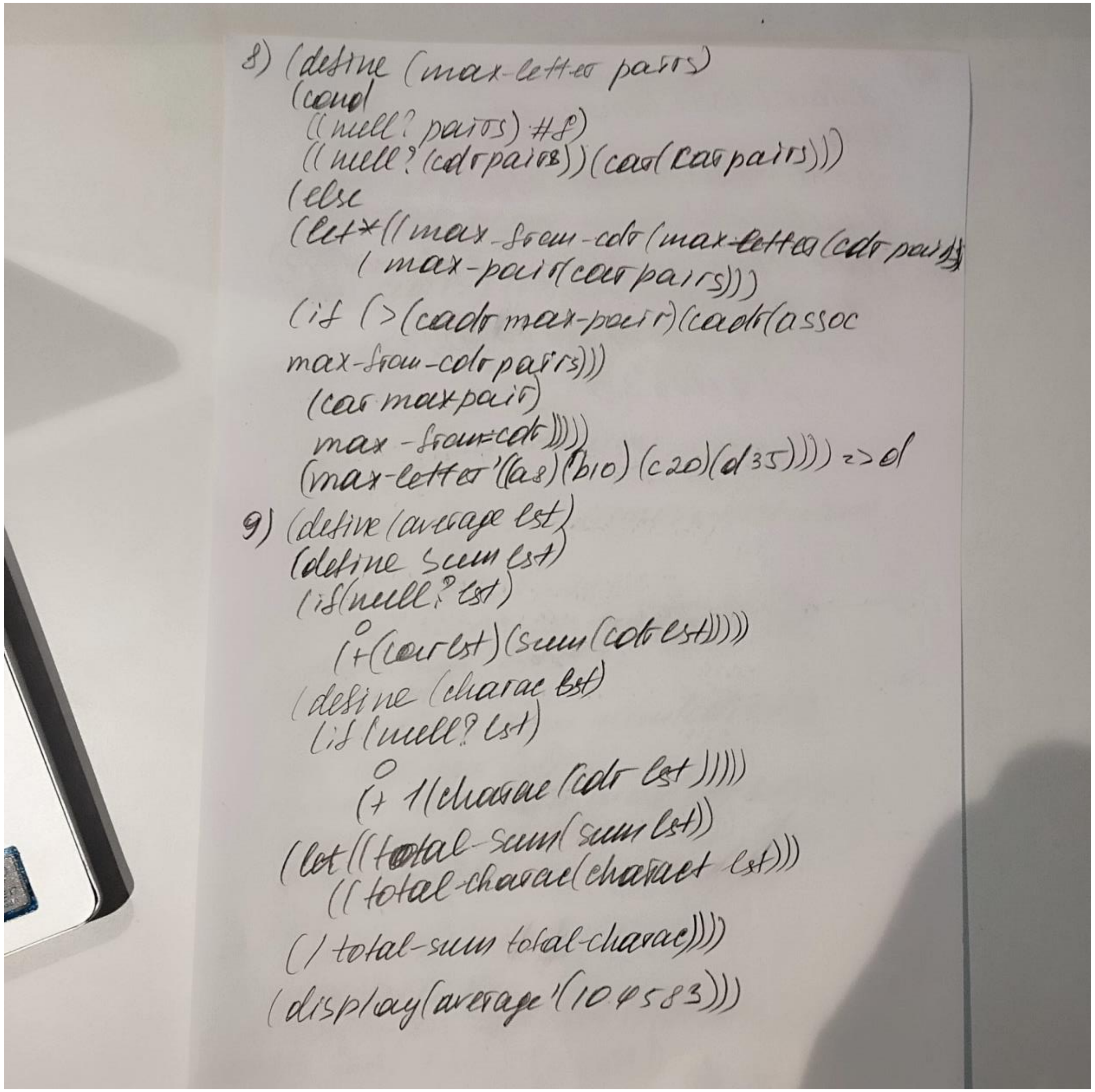


```
1) (define(replace arg1 arg2 lst)
    (cond
     ((null? lst)'())
     ((equal?(car lst) arg2)
     (cons arg1 (replace arg1 arg2(cdr lst))))

     (else
       (cons (car lst)(replace arg1 arg2)
       (cdr lst))))))
       (replace 1 3'(1 5 7 1 3))=>output(1 5 7 1 1)
2)  (define (replace-i-th-elem arg1 i list)
    (cond((null? lst)
    ((= i 0)
   (cons arg1(cdr lst)))
   (else
   (cons (car lst)(replace-ithelem arg1(- i 1)
   (cdr lst))))))
  (replace-i-th-elem 'x 3 '(abcd) outpud(abcx)
3) (define (count-occurs elem lst)
    (cond((null? lst)0)
   ((equal?(car lst)elem)
    (+1 (count-occurs elem (cdr lst))))
    (count-occurs 'a '(abaca))=>3
```



```
4) (define (remove_occur elem lst)
  (cond
    ((null? lst) '())
    ((equal? (car lst) elem)
     (remove_occur elem (cdr lst)))
  (else
  (cons (car lst) (remove_occur elem (cdr lst))))))
  (remove_occur 'a '(abaca) =s '(bc)
5) (difine (remove-first elem lst)
   (cond ((null? lst) '())
   ((equal? (car lst) elem)
   (cdr lst))
   (else (cons (car lst) (remove-first elem
    (cdr lst))))))
  (remove-first '6 '(67836)) => (7836)
6) (define (remove-less-than num lst)
   (cond ((null? lst) '())
   ((< (car lst) num)
   (remove-less-than num (cdr lst)))
   (else (cons (car lst)
  (remove-less-than num (cdr lst)))))))
 (remove-less-than 3 '(12345)); >(345)
7) (define (length-less? lstA lstB)
  (cond
  ((null? lstA) #t)
  ((null? lstB) #f)
 (else (length-less? (cdr lstA) (cdr lstB)))))
 (length-less? '(35) '(7268)) => #t
```



```
8) (define (max-letter pairs)
   (cond
    ((null? pairs) #f)
    ((null? (cdr pairs)) (car (car pairs)))
   (else
   (let* ((max-from-cdr (max-letter (cdr pairs)))
        ( max-pair (car pairs)))
   (if (> (cadr max-pair) (cadr (assoc
   max-from-cdr pairs)))
    (car max-pair)
    max -from-cdr)))))
   (max-letter '((a 8) (b 10) (c 20) (d 35)))) => d
9) (define (average lst)
   (define sum lst)
   (if (null? lst)
      0
      (+ (car lst) (sum (cdr lst)))))
   (define (charac lst)
    (if (null? lst)
       0
       (+ 1 (charac (cdr lst)))))
  (let ((total-sum (sum lst))
      ((total-charac (charact lst)))
  (/ total-sum total-charac))))
 (display (average '(10 4 5 8 3)))
```

10)

```
(define (insert element i lst)
(cond
  ((= i 0) (cons element lst))
 ((null? lst) '())
 (else (cons (car lst) (insert element (- i 1)
 (cdr lst))))))
(display (insert 11 1 '(7 4 9 38))) =>
      => (7 11 4 9 38)
```