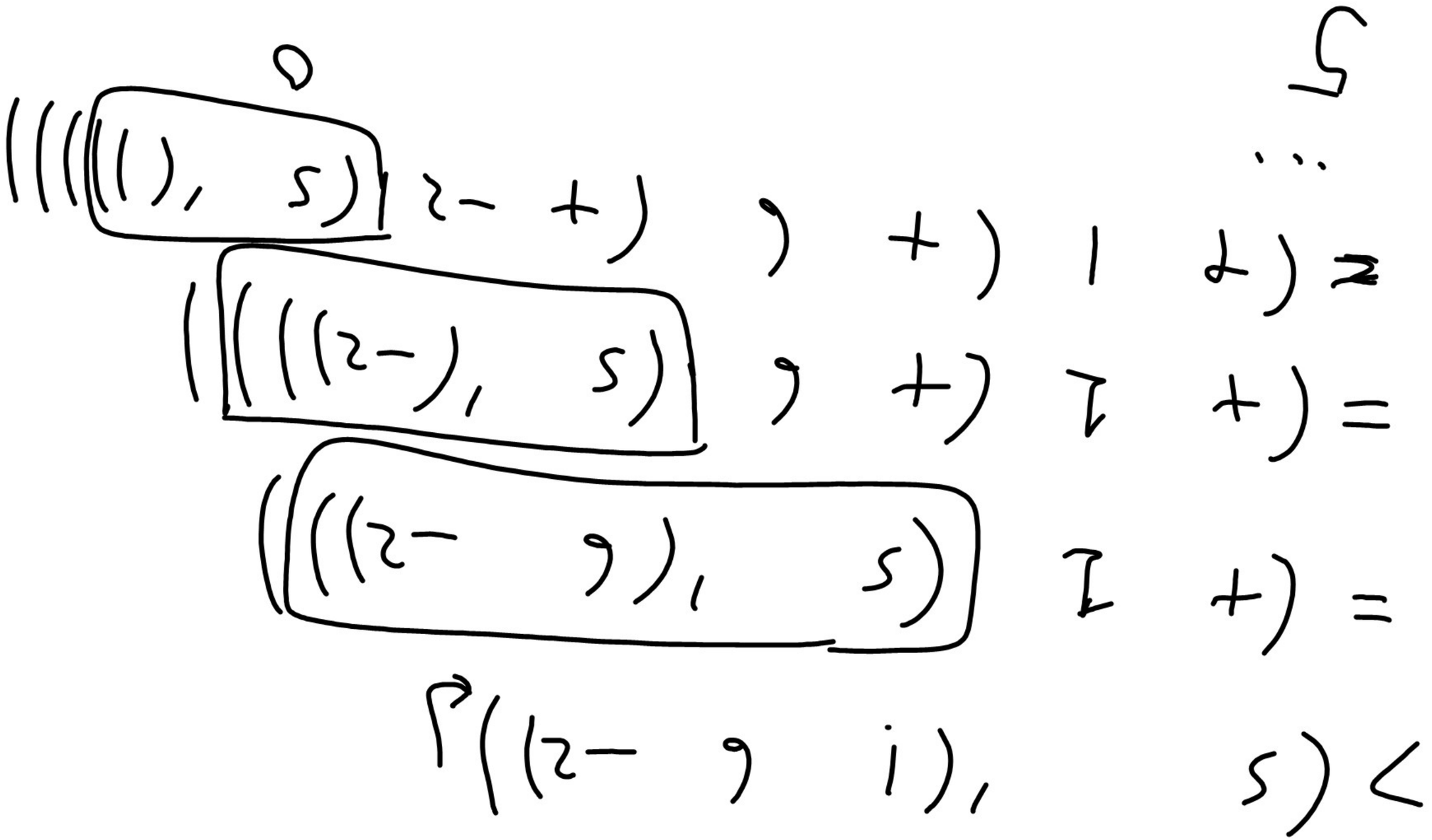
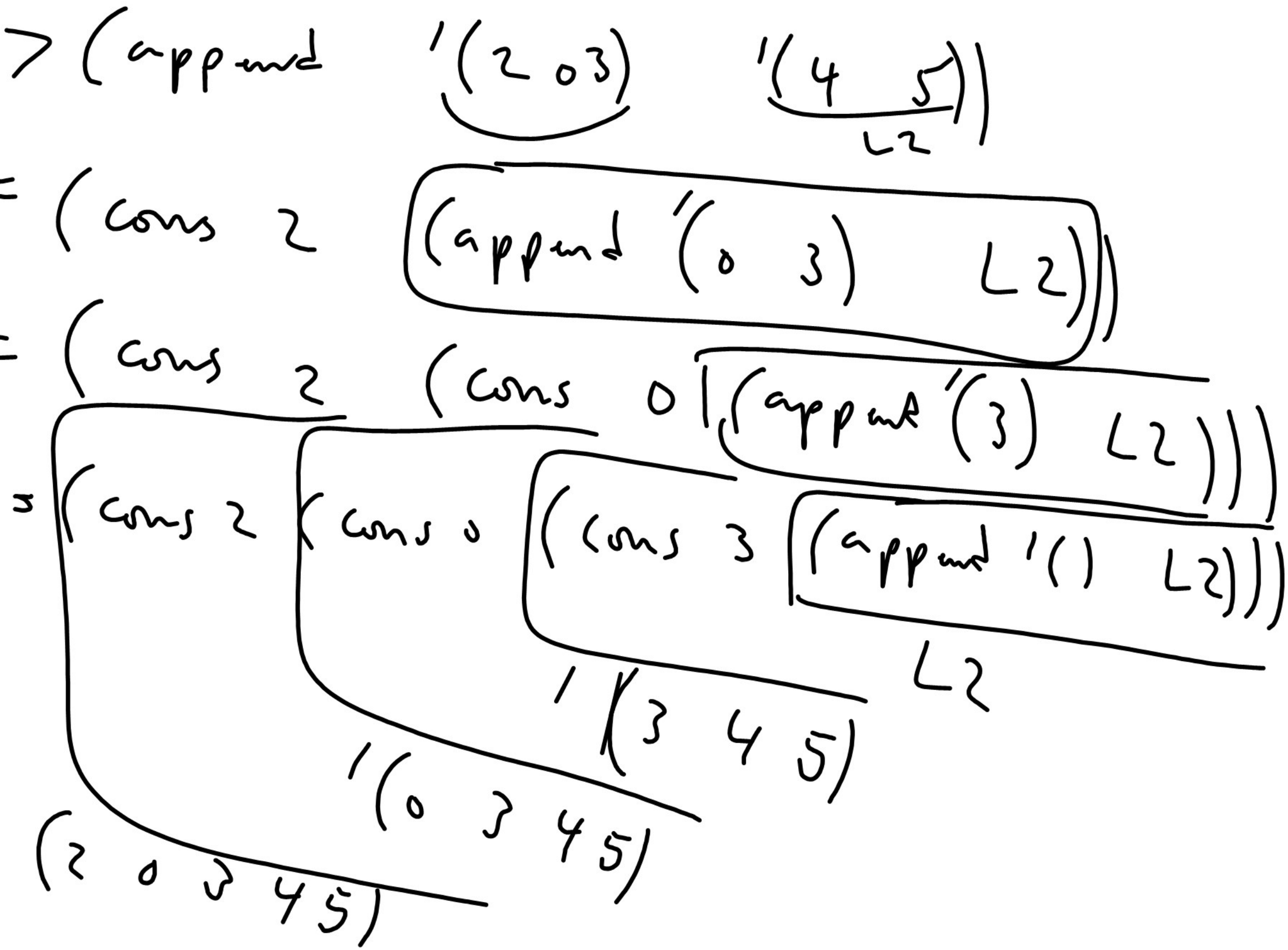


```
(define (abs-val x)
  (if (>= x 0) x (- x)))

(cond ((>= x 0) x)
      (else (- x)))
```


> (append '(2 0 3) '(4 5))
= (cons 2 (append '(0 3) L2))
= (cons 2 (cons 0 (append '(3) L2)))
= (cons 2 (cons 0 (cons 3 (append '() L2))))
(2 0 3 4 5)
1 (0 3 4 5)
1 (3 4 5)
L2
L2

Programming Languages

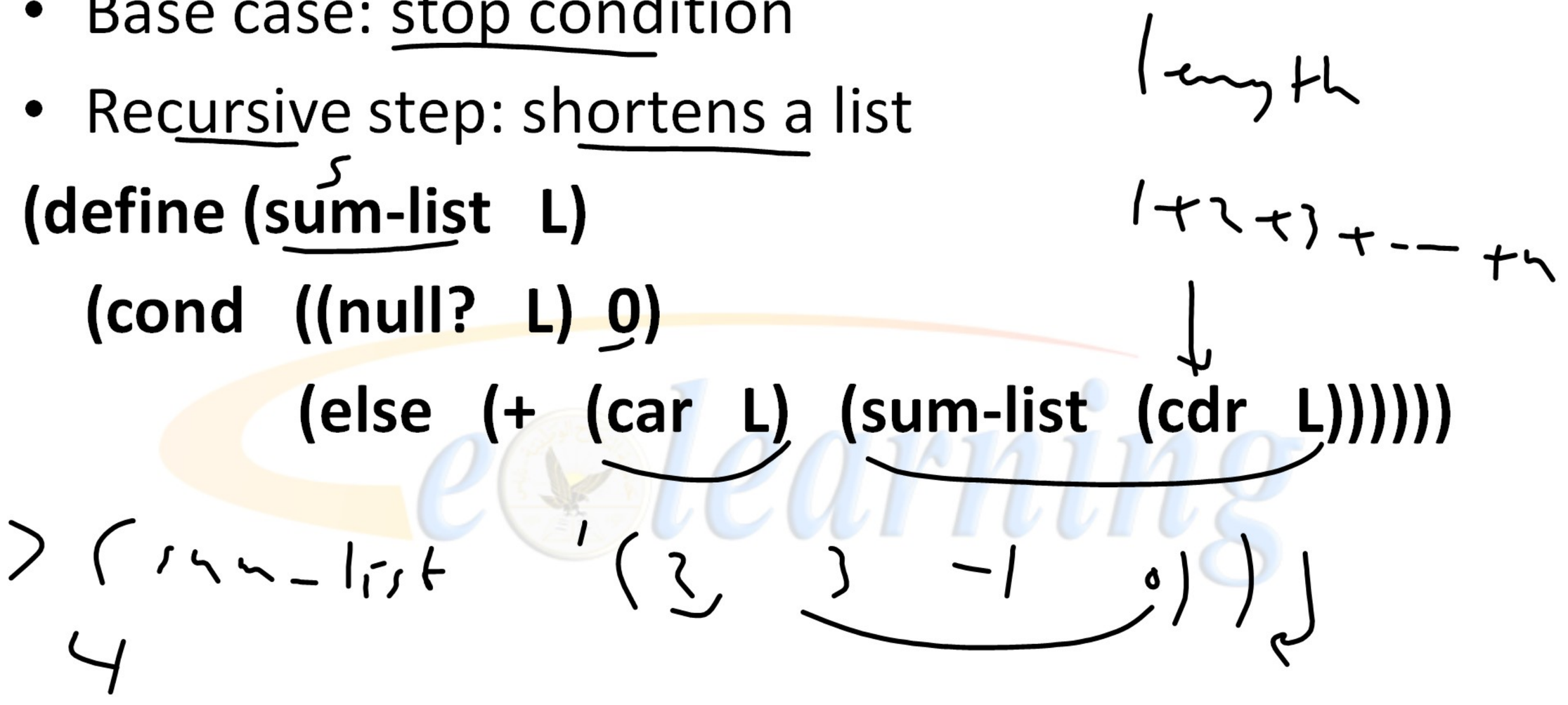
Wael Mustafa

Faculty of Engineering and Information Technology

Cdr recursion

- Base case: stop condition
- Recursive step: shortens a list

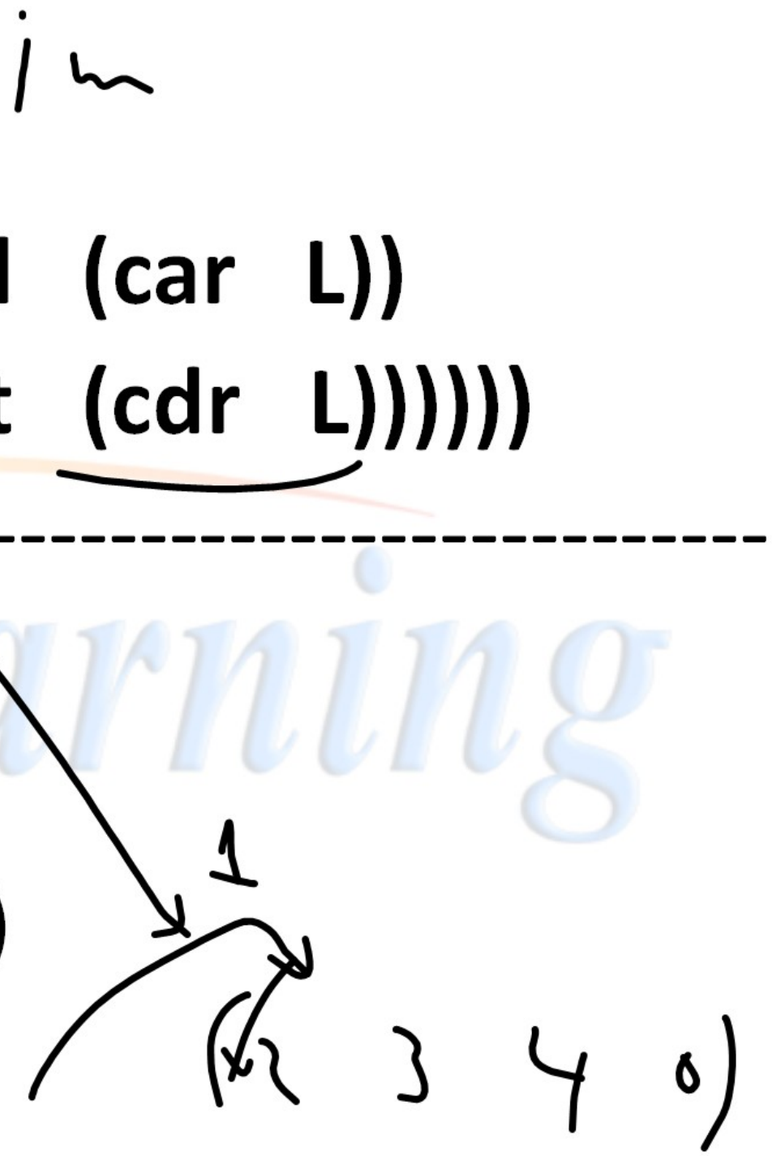
```
(define (sum-list   L)
   (cond   ((null?   L)  0)
           (else  (+  (car  L)  (sum-list  (cdr  L))))))
```



Cdr recursion: abs-list

```
(define (abs-list L)
   (cond ((null?  L)   '())
         (else (cons   (abs-val   (car   L))
                       (abs-list   (cdr   L))))))
```

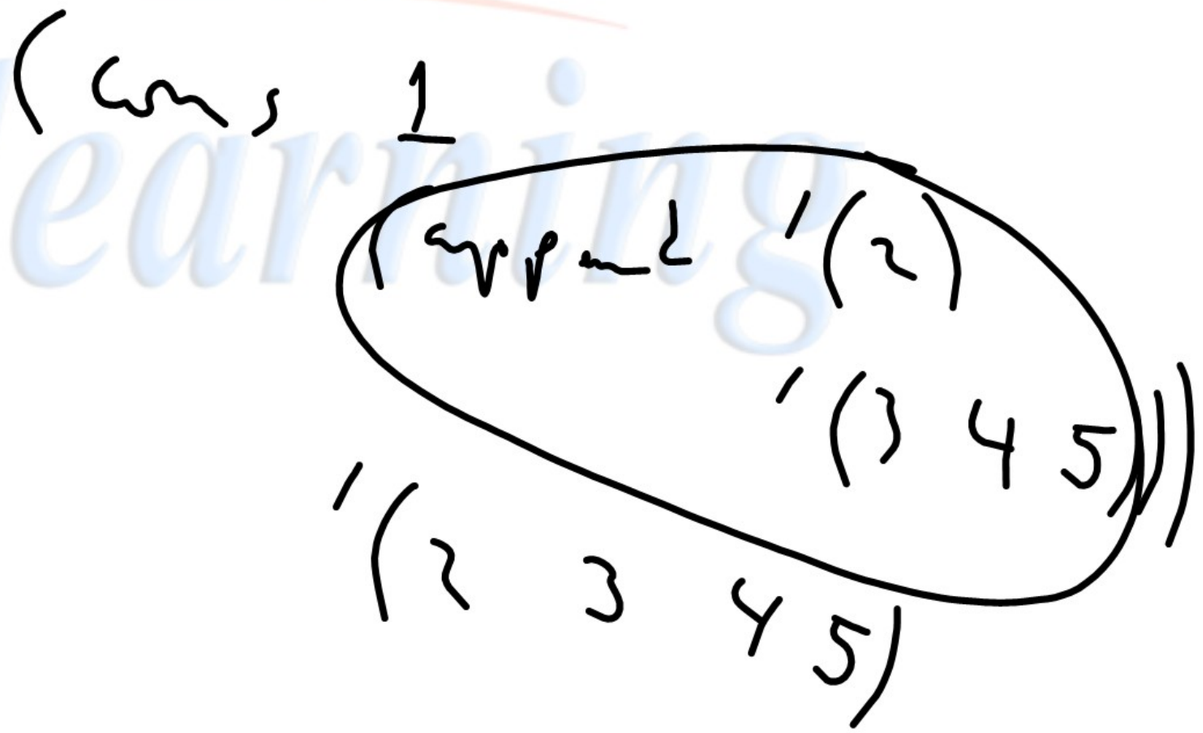
```
> (abs-list    '())
()
> (abs-list    '(1  -2   -3   4   0))
(1   2   3   4   0)
```



Cdr recursion: append

```
(define (append  L1   L2)
    (cond ((null? L1) L2)
          (else (cons  (car L1)
                       (append  (cdr L1)  L2)))))
```

```
> (append '(1 2)  '(3 4 5))
(1  2  3  4  5)
> (append '(1  2)  '(3  (4)  5))
(1   2   3   (4)   0)
> (append '( )  '(3  4  5))
(3  4  5)
> (append '(1  2)  '( ))
(1  2)
> (append  '( )  '( ))
( )
```



Car-cdr recursion

- If the first element is a list, then recursion on car- process nested levels
- Cdr recursion to advance the computation

```
(define (sum-list-nested   L)
   (cond   ((null?   L)  0)
           ((list?   (car   L))
            (+   (sum-list-nested   (car L))
                 (sum-list-nested   (cdr L))))
           (else   (+   (car  L)
                        (sum-list-nested   (cdr  L))))))
```

Car-cdr recursion: atomcount

```
(define (atomcount L)
   (cond ((null? L) 0)
         ((atom? L) 1)
          (else (+ (atomcount (car L))
                   (atomcount (cdr L))))))
```

> (atomcount '((1) (2) 3))

3

Problem: Evaluating the same expression twice

```
(define (longest-nonzero L1  L2)
   (cond  ((and (null?  L1)  (null?  L2))  -1)
          ((>  (length  L1)  (length  L2))  (length L1))
          (else  (length L2))))
```

What can be done without assignment statement?

Solution 1: Bind values to parameters in a helper function

```
(defime (maximum A B)
  (if (> A B) A B))
(define (longest-nonzero L1 L2)
  (cond ((and (null? L1) (null? L2)) -1)
        (else (maximum (length L1) (length L2)))))
```

Note: There is a built-in **max** function

Solution 2: Bind values to expression results using a let construct

```
(define (longest-nonzero L1   L2)
   (let   ((A   (length L1))
           ((B   (length L2))
    cond   ((and (null?  L1)  (null?  L2))  -1)
           ((>  A  B)  A)
           (else  B))))
```

Boolean operations

- And, or, not
- And, or evaluates as much as needed, similar to C
- And stops at the first false condition
- Or stops at first true condition

```
> (and (zero? 0)  (number?  2)  (eq?  1  1))
True
> (and (zero? 0)  (number?  'x)  (eq?  1  1))
False
> (or  (symbol?  'x)  (print  "yes"))
True
> (not (null? '(1  2)))
true
```

Equality checking for lists

```
(define (equal?  X  Y)
  (or (and  (atom?  X) (atom?  Y)  (eq? X  Y))
      (and (not  (atom?  X) (not  (atom?  Y))
           (equal?  (car  X) (car  Y))
           (equal?  (cdr  X) (cdr  Y)))))
```

```
> (equal?  'a  'a)
true
> (equal?  'a  'b)
false
> (equal?  '(a)  '(a))
```

Functions as parameters

```
(define (all-num? L)
        (or (null? L)
            (and (number? (car L))
                 (all-num? (cdr L)))))

(define (all-num-f f L)
   (cond ((all-num? L) (f L))
         (else 'error)))
```

```
> (all-num-f abs-list '(1 -2 3))
(1 2 3)
> (all-num-f abs-list '(1 a -3))
Error
> (all-num-f sum-list '(1 -2 3))
2
```

Functions as returned values

```
(define  (plus-list  x)
   (cond     ((number?  x)
                ((lambda  (y)  (+  (sum  x)  y)))
             ((list?  x)
                ((lambda  (y)  (+  (sum-list  x)  y)))
             (else  (lambda (z) z))))
```

closure

```
> ((plus-list   3)   4) ; execute closure on 4
10
> ((plus-list   '(1  3  5))   5); execute closure on 5
14
```

Functions as returned values: currying

```
(define pow
  (lambda  (x)
    (lambda  (y)
      (if   (=  y  0)
          1
          (*   x  ((pow   x)  (-  y  1)))))))
(define three-to-the   (pow  3))
(define  eightyone  (three-to-the 4))
(define sixteen   ((pow  2)   4))
```

Built-in higher-order functions: map

- map takes two arguments: a function and a list
- map builds a new list whose elements are the results of applying the function to each element of the input list

```
> (map abs '(-1 2 -3 4))
(1 2 3 4)
> (map (lambda (x) (+ 1 x)) '(-1 3 0))
(0 4 1)
```

Map function

```
(define (map  f   L)
  (cond   ((null?   L)   '( ))
          (else   (cons   (f   (car   L))
                          (map   f   (cdr   L))))))
```

Eval function

Evaluates an expression and returns the result

```
> (cons  '*   '(3   2    5))
(*   3    2  5)
> (eval  (cons  '*   '(3   2    5)))
30
> (eval   (list   '+    3   3   5))
11
> (eval  (append '(*   2  3)  '(4)))
16
> (eval  (cons  'append   '(   '(1  2)   '(3))  ))
(1   2   3)
```

What is wrong?

```
(define (atomcount s)
  (cond   ((null?   s)   0)
          ((atom? s)   1)
          (else   (+   (map   atomcount   s)))))
```

```
> (atomcount   '(a   b))
Run-time error: + expects <number> but given (1  1)
```

Correct atomcount using eval

```
(define  (atomcount  s)
  (cond  ((null?  s)  0)
         ((atom?  s)  1)
         (else  (eval  (cons  '+
                              (map  atomcount  s))))))
```

```
> (atomcount  '(a  b))
2
> (  atomcount  '((5) ((2 1)) (((7))))  )
4
```

High-order functions: apply

- Takes two arguments: a function and a list of arguments
- Executes the function on the elements of the list and returns the result

```
> (apply  +  '(3  3  5))
11
> (apply  append  '((7  3)  (5  4)))
(7  3  5  4)
```

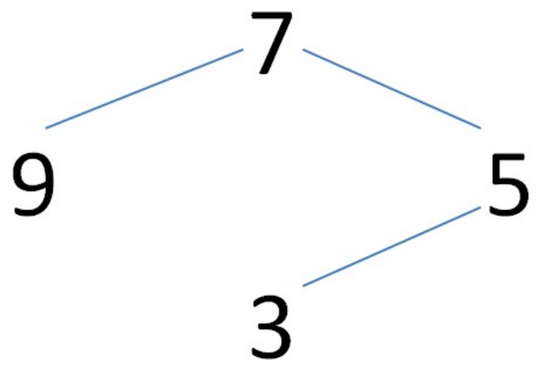
Correct atomcount using apply

```
(define (atomcount s)
  (cond ((null? s) 0)
        ((atom? s) 1)
        (else (apply + (map atomcount s)))))
```

Binary search trees in Racket

- Represent a tree as a list
- (root-element left-subtree right-subtree)
- Left-subtree and right-subtree are lists
- An empty tree is represented by an empty list ( )

Example



(7 (9 ( ) ( )) (5 (3 ( ) ( )) ( )))

Binary search tree insertion

```
> (insert  5  '())
(5  ()  ())
> (insert  5  '(9   (4  () ())  () ))
(9  (4  ()  (5  ()  ()))  ()  )
> (insert  50  '(9   (4  () ())  () ))
(9  (4  ()  ())  (50  ()  ()))
```

Binary search tree insertion

```
(define (key tree) (car tree))
(define (left tree) (cadr tree))
(define (right tree) (cadr tree))
```

Binary search tree insertion

```
(define (insert   el   tree)
   (cond  ((null? tree)  (list  el  ()  ()))
              ((= el  (key  tree))  tree)
              ((< el  (key  tree))
                   (list  (key tree)
                          (insert  el  (left   tree))
                          (right  tree)))
          (else    (list  (key tree)
                          (left  tree)
                          (insert  el  (right   tree))))))
```

Building a binary search tree from a list

- Given a list of numbers L, start from an empty BST and go through elements in L inserting each into the BST

```
(define (list-to-tree  L) (list-to-tree-helper  L  '()))
(define (list-to-tree-helper  L  tree)
    (cond  ((null?  L)  tree)
           (else  (list-to-tree-helper
                      (cdr  L)
                      (insert  (car L)  tree)))))
```

```
> (list-to-tree  '(5  3  8))
(5  (3  ()  ())  (8  ()  ()))
```