The way I understood Lisp.

R.Sreekumar, Ph.D.

July 14, 2020

1 Introduction. *[2020-07-08 Wed]*

I started learning SICP, but stopped in the middle because of lack of concentration and other parameters. However, that documentation is worth referring because, various things I tried to run a **scheme/lisp/racket** programs are documented there.

I had to install **Dr.Racket, mit-scheme, clisp** for my experimentation as well as **guile** and **geiser** for running my **scheme** programs within **org mode**. All these, and my experiments are documented there. The file is available as in this file in the same directory.

Since my work is spanning many things from LISP, Teaching, Programming, Software Development, Engineering Education documentation, this document will also not be concise.

2 Chapter 1: Building Abstractions with Procedures

Computational processes are abstract beings that inhabit computers. As they evalove, computational processes manipulate other abstract things called *data*. When the process evolve with a set of rules (or pattern of rules), it is called *program*.

2.1 Aside:

I wrote a program which returns boolean value. Generally lisp returns *true* or *false*. But this is only for primitive operations working on numbers such as equal to, not equal to, greater than, less than, etc.

This is my attempt that returning coustomized return values.

```
(define (divisible m n) (if (= 0 (remainder m n)) #t #f))
(list
   (divisible 4 8)
   (divisible 8 4))
```

3 Aside (From Little Schemer)

3.1 *[2020-07-11 Sat]*

I was working on **Little Schemer** examples. One of the problem was remove an element from list if it exisits. I tried my hand and the following is the code:

```
(define rember
  (lambda (a lat)
    (cond
      ((null? lat) ('()))
      (else
      (cond (eq? a (car lat)) (cdr lat)   ;; remove 'a' from 'lat'
       (else (rember a (cdr lat))))))))
```

To my surprise the code I have written was exactly same as the code given in book. This is because the two preivous programs **(lat? list)** and **(member? a lat)** from the book have similar pattern. Refer to this file.

However after seeing a seamless similarity between my program and author's program, I had a doubt. How it will maintain the previous list elements if the removed element is not the first one (the program uses **(eq? a (car lat)) => (cdr lat))**. Essentially, it will return the remaining elements of the list excluding the current element. Looks good. What about the previous elements in the list? I had this doubt and thought that since it is a recursive function, the remaining elements will be automatically appended. To convince my answer, I wrote the workdown of the entire process in two different ways. They are (1-13: first way, 15-20: second way):

```
1  (rember 'team '(I am a team member))
2  (null? lat) => #f
3  (eq? 'team 'I) => #f
4     (rember 'team '(am a team member))
5     (eq? 'team 'am) => #f
6        (rember 'team '(a team member))
7        (eq? 'team 'a) => #f
```

```
 8          (rember 'team '(team member))  ;; retruns (member)
 9          (eq? 'team 'team) => #t => (member)
10             (rember 'team 'member))
11             (eq? 'team 'member) => #f
12               (rember 'team '())  ;; (cdr lat) is empty list.
13               (null? '()) => #t => returns '()
14
15  (rember 'team '()) => '()  ;; terminating condition
16  (rember 'team '(member) => '()
17  (rember 'team '(team member)) => '(member)
18  (rember 'team '(a team member)) => '(a member)
19  (rember 'team '(am a team member)) => '(am a member)
20  (rember 'team '(I am a team memeber)) => '(I am a membber)

 In other words, each successive recursive call returns a
 replacement list for the input list.
```

The above workout is good and I had to convince myself with the power of recursion. But the fact is that only two lines (line number 8 and 12) are only correct. Essentially it follows that the end product will be only an empty list '(). Again this is my assumption after reading the book again.

The book tells that the elements before the term matched are left out and we have no mechanism to keep them up. The book is trying to introduce **cons** in a different way.

Understanding the above problem, I tried to execute my code in **Dr. Racket**. This is the output.

```
> (rember 'team '(I am a team memeber))
'I
```

The output was a total surprise for me. It is neither empty list nor the list with (member) as I expected.

I am yet to fix the bug in the above code.

I wrote another code exactly similar but with one difference is replacing **'()** with **(quote ())** (as given in the book).

The code as written in the "Dr. Racket IDE":

```
> (define eliminate
    (lambda (a lat)
      (cond
```

```
        ((null? lat) (quote()))
        (else (cond
                ((eq? a (car lat))(cdr lat))
                (else (eliminate a (cdr lat))))))))
> (eliminate 'team '(I am a team player))
'(player)
> (eliminate 'a '(I am a team player))
'(team player)
> (eliminate 'I '(I am a team player))
'(am a team player)
>
```

And the previous code is as in "Dr. Racket":

```
> (define rember
     (lambda (a lat)
       (cond
         ((null? lat) (quote ()))
         (else
         (cond (eq? a (car lat)) (cdr lat)
          (else (rember a (cdr lat))))))))
> (rember 'team '(I am a team player))
'I
> (rember 'a '(I am a team player))
'I
> (rember 'I '(I am a team player))
'I
>
```

The first code I have written and analyzed is a buggy one becuase of paranthesis. `(eq? a (car lat)) (cdr lst)`. The `<stmt>` to be executed lies outside `<condition >`. The proper form is `(<cond> <stmt>)`. Hope, I won't do this mistake again.

The correct code is:

```
(define rember
  (lambda (a lat)
    (cond
      ((null? lat) (quote ()))
      (else (cond
```

```
            ((eq? a (car lat)) (cdr lat))
            (else (rember a (cdr lat))))))))
```

Now we will go for the cons function. The previous code is fine but the problem is that atoms found before the atom we want replace are left out. The final result will have only atoms after that.

Examples:

```
(rember 'team '(I am a team player))
=> '(player)
(rember 'am '(I am a team player))
=> '(a team player)
```

Notice that the terms before the search terms are left out. These has to be appended to the original list. This can be achieved by using **cons** construct.

Though I had some idea about **cons**, I was not very clear about recursion. I was entering into endless loop. The reason is that my code is:

```
(define rember
  (lambda (a lat)
    (cond
      ((null? lat) (quote ()))
      (else (cond
              ((eq? a (car lat)) (cdr lat))
              (else (rember (cons (car lat) (cdr lat)))))))))
```

Notice that last line.

If **(car lat)** is not **a**, then I am recursively calling **rember** with the element **(cons (car lat))** appended to **(cdr lat)**. So in case of **I am a team memebr**, since **I** is not the term to be repalced (that is **team**), **I** is appended to the list and hence becomes **I am a team player**, instead of **am a team player**. Since, this is happening recursively, the **list** will never become **null** and will never get terminated.

Here is the final code.

```
(define rember
  (lambda (a lat)
    (cond
      ((null? lat) (quote ()))
      (else (cond
              ((eq? a (car lat)) (cdr lat))
              (else (cons (car lat) (rember a (cdr lat)))))))))
```

In the above code, we can see the power of **recursion**. Before appending **(car lat)**, we invoke **(rember a (cdr last))** hence **(car lat)** waits until the return value of last **(rember a (cdr lat))** which is **(member)**. Then it reverts back to append each **atom** at each stage of **back recursion**.

A quote from the book:

> But since we don 't know the value of (rember a (cdr lat)) yet, we must find it before we can cons (car lat) onto it.
>
> "The function rember checked each atom of the lat , one at a time, to see if it was the same as the atom and. If the car was not the same as the atom , we saved it to be consed to the final value later. When rember found the atom and, it dropped it, and consed the previous atoms back onto the rest of the lat."

We can actually remove two **cond** in the above code (again from book) so that the code looks elegant.

```
(define rember
  (lambda (a lat)
    (cond
      ((null? lat) (quote ()))
      ((eq? a (car lat)) (cdr lat))
      (else (cons (car lat) (rember a (cdr lat)))))))
```

Essentially, **cond** can do for multiple checks. It is checking for:

1. Whether the list is null, do something
2. Whether the atom is equal to search atom, do something
3. Else do something (recursively).

Moving on, the next problem is collect all the first S-expressions from a list that consists of **only lists**.

We shall see an example (ofcourse, there are 3 differect test cases):

```
Given Input:
((one two) (three four) (five six))
Expected Output:
(one three five)
```

I wrote the program after little bit difficulty. However, I got the answer. My code is:

```
(define firsts
 (lambda (l)
   (cond
     ((null? l) (quote()))
     (else (cons (car (car l)) (firsts (cdr l)))))))
```

Let me check the answer from the book. It is exactly same as I have written but with more explanation. That means I can skip the next few pages.

The next problem is to repace an old atom with new atom in a list. Let me think. See you tomorrow.

3.2 *[2020-07-12 Sun]*

It turned out to be too trivial. I wrote the first program to replace 'old with 'new in a "list".

The program is:

```
(define insertR1
  (lambda (old new  lat)
    (cond
      ((null? lat) (quote ()))
      (else (cond
              ((eq? old (car lat)) (cons new (cdr lat)))
              (else (cons (car lat) (insertR1 old new (cdr lat)))))))))
```

This will result in:

```
Given old = 'fudge, new = 'topping and
lat = '(ice cream with fudge for dessert), the program
will give,
'(ice cream with topping for dessert)
```

The book wants also the old character along with new character. I guess, that is why the name **insertR**. The new code is:

```
(define insertR
  (lambda (old new  lat)
    (cond
      ((null? lat) (quote ()))
      (else (cond
              ((eq? old (car lat)) (cons old (cons new (cdr lat))))
              (else (cons (car lat) (insertR old new (cdr lat)))))))))
```

This will result in:

```
Given old = 'fudge, new = 'topping and
lat = '(ice cream with fudge for dessert), the program
will give,
'(ice cream with fudge topping for dessert)
```

The above code is fairly simple. Or I am getting the hang of lisp.

There are two more addition to this. Inserting to the left and substitution. Substitution is trivial and the code is exactly the same as **insertR1** shown above.

The left appending can be done in two ways:

```
(cons new (cons old (cdr lat)))
or
(cons new lat)
```

The second one is more elegant and intutive.

There is a code substituting which we have done already. There is a subst2 method which substitute a "new" value for either the occurance of "old1" or "old2" values. This fairly easy. But a point to not that comparison can be done in two ways.

```
((eq? (car lat) o1 ) (cons new (cdr lat)))
((eq? (car lat) o2) (cons new (cdr lat)))

 or

(or (eq? (car lat) o1 ) (eq? (car lat) o2))
```

The above is to illustrate the use of **or** in lisp. What we want to know is that, the same thing can be done by using

```
(cond ((<test1>)(<ans1>))
      ((<test2>)(<ans2>))
      ((<test3>)(<ans3>))
      (else <ans4>))
```

To put it in another way, **cond** responses to multiple tests in a secquential fashion. This is similar to **swith-case** statements available in **c** or **Java**.

The code for **multirember** turned out to be very simple. It is same code for **rember** but when the **atom** is the one to be removed, we pass **(multirember a (cdr lat))**. Here is the code:

```
(define multirember
  (lambda (a lat)
    (cond
      ((null? lat) (quote ()))
      (else (cond
         ((eq? a (car lat)) (multirember a (cdr lat)))
         (else (cons (car lat) (multirember a (cdr lat))))))))))
```

Added few more *primitive operations* or *special forms* while watching the vedio "**Scheme, Guile, and Racket: an Introduction by Craig Maloney**".

Keep an eye on this website. It is from **University of taxas in Austin**. Again, **scheme** in a differnet way.

The "How to Design Programs" is a good read. But it is only web based and currently 2nd edition is available, which was released in 2018.

I had a bookmark in YouTube for CS510 conducted by a Proffesor teaching Racket.

3.3 *[2020-07-13 Mon]* (C-c !) Two versions of multiinsertR

NOTE: The discussion below is unnecessary. It was a typo I had in the program which caused the prolem.

Yesterday, I was looking at **multiinsert**. Refer previous section for **insertR** (insert the new atom to the right of the item we are searching for) and **multirember** for removing multiple occurance of an 'atom' in a 'list'.

Based on these observations, I wrote **multinsertR** assuming the same patterns. I have two verions. The reason for documenting these versions is that I am getting weired results which are not at all acceptable.

Version 2 is obviously non-terminating recursion as **multiinsert** is called again and again with newly appended list which never terminates. But Version 1 need to be seen to figure out where the problem is.

3.3.1 Version 1

This version is a blending of the preivious two methods.

```
(define multiinsertR
  (lambda (old new  lat)
    (cond
      ((null? lat) (quote ()))
       (else (cond
          ((eq? old (car lat))
```

```
            (cons old (cons new
                  (multiinsertR old new (cdr lat)))))
        (else (cons (car lat)
              (insertR old new (cdr lat)))))))))
```

You can see the **typo** in the last line. Instead of calling **multiinsertR**, I am calling **insertR** which is a previously defined function. Obviously the result will be unexpected.

3.3.2 Version 2

This version doesn't require any discussion as the 'list' will never become '() and hence never terminate.

```
(define multiinsertR
  (lambda (old new  lat)
    (cond
      ((null? lat) (quote ()))
        (else (cond
          ((eq? old (car lat))
             (multiinsertR old new
                (cons old (cons new (cdr lat)))))
          (else (cons (car lat)
                (insertR old new (cdr lat)))))))))
```

The whole topic so far is useless, but for a beginner these code will give more insight into recursive functions.

3.3.3 Version 3: At last the correct version.

Now for the correct one:

```
(define multiinsertR
  (lambda (old new  lat)
    (cond
      ((null? lat) (quote ()))
       (else (cond
          ((eq? old (car lat))
               (cons old (cons new
                     (multiinsertR old new (cdr lat)))))
          (else (cons (car lat)
                (multiinsertR old new (cdr lat)))))))))
```

3.3.4 Now for multiiinsertL.

I am copying the code from book. It says it will never terminate.

```
(define multiinsertL
   (lambda (new old lat)
     (cond
      ((null? lat) (quote ()))
      (else
       (cond
        ((eq? old (car lat))
         (cons new (cons old (multiinsertL new old lat))))
        (else
         (cons (car lat (multiinsertL new old (cdr lat)))))))))))
```

And the way it is executed in Dr.Racket.

```
> (multiinsertR 'one 'ten l1)
'(one ten two one ten two one ten two)
> (load "/home/shree/scip/sicp/little-schemer.scm")
> (define l2 '(chips and fish or fish and fried))
> (multiinsertL 'fried 'fish l2)

<< infinte loop>>
```

So the task for the day is figure out why it is not terminating (the obious answer is the 'list' is not becoming null) and figure out how to overcome it.

Here we start tracing:

```
(define new 'fried)
(define old 'fish)
(define lat '(chips and fish or fish and fried))
;; mil is short form for multiinsertL
(mil 'fried 'fish lat)
(?eq 'chips 'fish) =>#f
(cons 'chips (mil 'fried 'fish lat)
(?eq 'and 'fish) =>#f
(cons 'and (mil 'fried 'fish lat)
(?eq 'fish 'fish) => #t
(cons 'fried (cons 'fish (mil new old lat)))

<<
```

```
Yes, I understand.  At this step the value of lat is:
'(fish or fish and fried)
We are passing it again, and it matches with 'fish and
goes into infinite recursion.
>>
```

I gues, I found the reason for infinite recursion. This happens because

```
'(chips and fish ....
should give
'(chips and fried fish  ...
```

I guess, we can overcome this problem using (cdr (cdr lat)). Let me check. This is my version of code. To avoid fish, I am doing two cdr to passing list.

```
(define multiinsertL
 (lambda (new old lat)
   (cond
    ((null? lat) (quote ()))
    (else
     (cond
      ((eq? old (car lat))
       (cons new (cons old (multiinsertL new old (cdr (cdr lat))))))
      (else
       (cons (car lat) (multiinsertL new old (cdr lat))))))))))
```

And the output is:

```
> (multiinsertL new old lat)
'(chips and fried fish fried fish fried)
> lat
'(chips and fish or fish and fried)
```

Obviously my code is wrong as 'and' and 'or' are ommitted. We are close to the solution.

Is it possible to use

```
(cons new (cons (multiinsertL new old (cdr lat)) old))
```

My second version is give below:

```
(define multiinsertL
  (lambda (new old lat)
    (cond
```

```
     ((null? lat) (quote ()))
     (else
      (cond
       ((eq? old (car lat))
        (cons new (cons (multiinsertL new old (cdr lat)) old)))
       (else
        (cons (car lat) (multiinsertL new old (cdr lat)))))))))
```

Yup. I have modified. Let's see the result.

```
> lat
'(chips and fish or fish and fried)
> (multiinsertL new old lat)
'(chips and fried (or fried (and fried) . fish) . fish)
```

Wonderful result. Though it is wrong, there are many things to learn. That's why I documented it.

Let's check to original version **insertL** before replacing two 'cars' with a single 'cons'.

Bringing the idea from eariler, we have

```
(cons new (cons old (cdr lat)))
or
(cons new lat)
```

It worked. And the final version is:

```
> (define multiinsertL
    (lambda (new old lat)
      (cond
       ((null? lat) (quote ()))
       (else
        (cond
         ((eq? old (car lat))
          (cons new (cons old (multiinsertL new old (cdr lat)))))
         (else
          (cons (car lat) (multiinsertL new old (cdr lat))))))))))
  > lat
'(chips and fish or fish and fried)
> (multiinsertL new old lat)
'(chips and fried fish or fried fish and fried)
```

Since reaching so far, **multisubst** becomes so trivial. Here is the code:

```
(define multisubst
  (lambda (new old lat)
    (cond
     ((null? lat) (quote ()))
     (else
      (cond
       ((eq? old (car lat))
        (cons new (multisubst new old (cdr lat))))
       (else
        (cons (car lat) (multisubst new old (cdr lat))))))))))
```

3.3.5 Working with numbers (Chapter 4)

It looks easy as numbers and their manipulation are done by prefix notation. I am comfertable with these.

```
(+ 3 5)
(* 3 (- 10 5)
etc.
```

But book starts this chapter slightly different. They define tuple as

```
'(1 2 3 4 5)
```

which is a list consisting of only numbers. If it has another type of element like

```
'(1 2 3 cool 4 5)
```

it is no more tuple.

The `(null? list)` test instead returning an `()` or empty list, returns 0. Hence the comparison operator is written as:

```
((null? lat) 0)
```

The first program is to add the elements in list or tuple. The code is trivial:

```
(define addtub
  (lambda (lat)
    (cond
     ((null? lat) 0)
     (else (+ (car lat) (addtub (cdr lat)))))))
```

And there is no need for explantion.

This chapter has **natural recursion** way. To understand that we need a way add 1 and subtract 1 from a number. This is trivial. The way I have written is

```
;; i++; lol
(define (add1 x)
  (+ x 1))

;; i--; lol
(define (sub1 x)
  (- x 1))
```

The book has lambda notation inbuilt like previous programs.

They are:

```
(define add1
   (lambda (n)
      (+ n 1)))

(define sub1
   (lambda (n)
      (- n 1)))
```

These code are used for introducinng multiplication and division recursively. We have m x n. One way of multiplying it is adding m to itself while n is reducing to 0. Hope, this will make you understand that how it can be done. The code is:

```
(define multi
  (lambda (m n)
    (cond
     ((zero? n) m)
     (else (+ m (multi m (sub1 n)))))))
```

I have goofed up somewhere. I am not getting the expected result. It is:

```
> (multi 3 2)
9
```

It is multiplying m to itself n times (m^n). This is power of m to n. We need multiplication.

Ok. I understood the mistake. I should return 0 when n = 0, that means don't add it again.

Now the code looks like:

```
(define multi
  (lambda (m n)
    (cond
     ((zero? n) 0)
     (else (+ m (multi m (sub1 n)))))))
```

Knowing this division will be similar.

I think I am loosing interst and almost all the problems are similar and using **natural recursion**. I have enough of it. May be go through the book and document if anything difficult is found.

There are two similar examples. The first one, I am just typing as they have given, while the second one, I am going to try.

First one is to find the length of a tuple.

```
(define length
  (lambda (lat)
    (cond
     ((null? lat) 0)
     (else (add1 (length (cdr lat)))))))
```

No discussion on the above code as it is self explantory. I want to add one point here.

```
in mit-scheme
(define l '(1 2 3 4 5))
in racket and lisp
(define l (list 1 2 3 4 5))
```

The second one to tell the position of an atom in the list. Given a list with `(Mary had a little lamp)` and asking for `little`, the function should return 4.

Here is my try:

```
(define position
  (lambda (x lat)
    (cond
     ((null? lat) 0)
     ((eq? x (car lat)) (add1 0))
      (else (add1 (position x (cdr lat)))))))
```

Surprisingly this program works. The only error is that if the 'atom is not in the 'list, it will return max value. You can refer to the below snippet.

```
> ml
'(mary had a little lamp)
> (positon 'marry ml)
5
> (postion 'mary ml)
1
```

Let's see how the book is solving the problem. It is accepting another parameter for **pick**. That is **n**, which gives the position.

I guess, I misunderstood the problem. The **pick** is essentially to return the atom at that position. So I am renaming my code in the previous example.

Now restating the problem, given `(mary had a little lamp)` and `n = 4`, the function should return `little`.

Here is code written by before seeing the book.

```
(define pick
  (lambda (n lat)
    (cond
     ((null? lat) 0)
     ((eq? 1 n) (car lat))
     (else (pick (sub1 n) (cdr lat))))))
```

and the output is:

```
> ml
'(mary had a little lamp)
> (pick 2 ml)
'had
> (pick 6 ml) ;; out of bound
0
```

The code in the book is slightly differnt:

```
(define pick
   (lambda ( n lat)
      (cond
        ((zero ? (sub1 n)) (car lat) )
        (else (pick (sub1 n) (cdr lat))))))
```

It looks slightly elegant by using `(zero? (sub1 n))` construct instead of `(eq? 1 n)` construct. I agree. One of the 10 comandments is test `(null? lat)` for lists and test `(zero? num)` for numbers.

So, I have written two functions purely by me. The first one is my invention. If it is there in the book at a latter point of time, it will document it just for comparing with my code.

There are few more assignment is **numbers** before going to next chapter.

3.4 *[2020-07-14 Tue]* Continuing with numbers

3.4.1 More and more numbers

Continuing with **pick**, we are going to write **rempick**, which stands for removing the picked element from the list. Example:

```
lat = (hotdogs with hot mustard)
n = 3
(rempick n lat) => (hotdogs with mustard)
```

I guess, I can modifiy my **pick** code.

```
(define rempick
   (lambda ( n lat)
     (cond
        ((null? lat) (quote ()))
        ((zero? (sub1 n)) (rempick n (cdr lat)) )
        (else (cons (car lat) (rempick (sub1 n) (cdr lat)))))))
```

According to my logic:

1. Check the element. If it is not in the position a. Cons it to the list b. Recursively call the **rempick** with (cdr lat)
2. If the element is in the position a. recursively call with (cdr lat)
3. If the list is empty return '().

But the output is:

```
> lat
'(hotdogs with hot mustard)
> (rempick 4 lat)
```

```
'(hotdogs with hot)
> (rempick 3 lat)
'(hotdogs with)
> (rempick 2 lat)
'(hotdogs)
>
```

This shows clearly we are eliminating the tail part after the position.

I have found the solution in two iterations. The first one returning 'lat' when the element to be removed is found and the second one returning the '(cdr lat)' when the element is to be found.

Note, as in the previous code, we are not recursively calling `(rempick n (cdr lat))`.

I am copying my work as I did it in Racket IDE here. You can identify the mistake and see the results of that mistake.

```
> ;; version 1 (returning lat)
> (define rempick
   (lambda ( n lat)
     (cond
        ((null? lat) (quote ()))
        ((zero? (sub1 n)) lat )
        (else (cons (car lat) (rempick (sub1 n) (cdr lat)))))))

> lat
'(hotdogs with hot mustard)
> (rempick 3 lat)
'(hotdogs with hot mustard)
> (rempick 2 lat)
'(hotdogs with hot mustard)
> (rempick 1 lat)
'(hotdogs with hot mustard)

;; version 2 (returning (cdr lat)
> (define rempick
   (lambda ( n lat)
     (cond
        ((null? lat) (quote ()))
        ((zero? (sub1 n)) (cdr lat) )
        (else (cons (car lat) (rempick (sub1 n) (cdr lat)))))))
```

```
> lat
'(hotdogs with hot mustard)
> (rempick 3 lat)
'(hotdogs with mustard)
> (rempick 2 lat)
'(hotdogs hot mustard)
> (rempick 1 lat)
'(with hot mustard)
>
```

There is a program to find whether the given atom is a number or not. This method `(number? x)` is already available with **scheme**. However, even if it is not there, we can write it as scheme gives opportunity. How?

I write my own version.

My version is:

```
(define mynum
  (lambda (x)
    (cond
       ((and (atom? x) (not (string? x))) #t)
       (else #f))))
```

It is a very trivial program, but not that I am using another primitive `(string? x)`.

The book says we cannot write `(number? x)` as it a primitive function like `(atom? x)`, `(null? x)` `(cons x lat)` etc.

The function **no-num** returns a list without numbers when given a list mixed with numbers and strings.

I think no need to write this function, as it is very similar to previous one we did. In that function, we removed the element at the position n. Here we need to check each element whether it is a number or not. If it is a number, skip it with simply (cdr lat). It will do the things.

I guess all the other functions are a kind of similarity.

1. Function `(all-nums lat)` returns only numbers instead of strings.
2. Function `(eqan a1 a2)` returns true if both the atoms are same. In case of number we check with `(= a1 a2)` and in case of strings we check with `(?eq a1 a2)`.
3. Function `(occur a lat)` returns the number of times `a` occuring in `lat`. This is a mixture of `position` and `rempick` in a slightly different way.

4. Function `(rempick n lat)` can be rewritten by using `(one? x)` question instead of `(zero? x)`. Remember we have solved the `position` problem using 1 first and then looked in to the book to see that it can be solved by using 0. The same has been repeated here.

3.4.2 "Oh My Gwad", (Chapater 6)

This chapter deals with special notation '*', representing 'all'. For example, our first method is `(rember* a lat)` means remove all occurances of **a** in **lat**.

As usual we will try our hand (many times) and if there is no other way, we will look into the book.

In lisp fashion,

```
(cond
   ((ans? (try method)) (write answer))
   ((?not-satisfied) (try another method))
   (else (look into book)))
```

Here goes my program. That also tells that I am yet become strong. The code is:

```
(define rember*
  (lambda (x lat)
    (cond
     ((null? lat) (quote ()))
     ((eq? (car lat))(rember* x (cdr lat)))
     (else (cons (car lat) (rember* x (cdr lat)))))))
```

And the output in racket is:

```
> (define lat '((coffee) cup ((tea) cup) (and (hick)) cup))
> (define x 'cup)
> (rember* x lat)
'((coffee) ((tea) cup) (and (hick)))
```

What mistake we have done? The only mistake, I guess is we are not going into the list within lists.

I have modified my program at least twice, and it looks like this now:

```
(define rember*
  (lambda (x lat)
```

```
(cond
 ((null? lat) (quote ()))
 ((lat? (car lat)) (rember* x (car lat)))
 ((lat? (cdr lat)) (rember* x (cdr lat)))
 ((eq? x (car lat)) (rember* x (cdr lat)))
 (else (cons (car lat) (rember* x (cdr lat))))))
```

I did a small change in program by replacing the first **lat** condition namely `((lat? (car lat)) (rember* x car lat))` to `((pair? (car lat)) (car lat) (rember* x (cdr lat)))`.

So some points here from my **trails** in the Racket IDE.

1. **(lat? lat)** (the program we have writen earlier) will check only a list with atoms and not with list with lists.

   ```
   Examples:
   (lat? '(1 2 3 4)) => #t
   (lat? '(1 (2 3) 4) => #f
   ```

   (Note: We used `lat?` in **multirember** function. In that case, we had only multiple atoms in the list. In case of **rember***, we need to check whether the atom is present inside a list of a given list).

2. **(pair? lat)** (built-in primitive) checks whether an element is a list even with one element.

   ```
   Example:
   (pair? '((coffee))) => #t
   (pair? '(coffee)) => #t
   (pair? 'coffee) => #f
   ```