Four operations on the list: car, cdr, null?, cons

```
(car '(a b c)) => a
(cdr '(a b c)) => (b c)
```

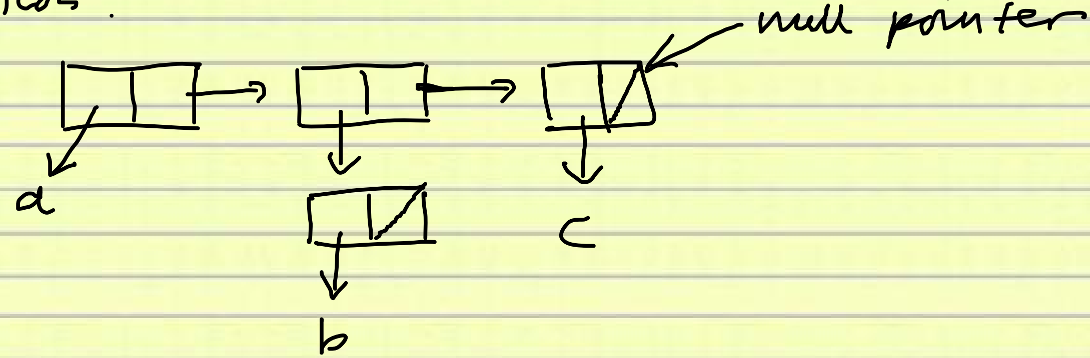
Neither car nor cdr change the list they operate on.

e.g.

```
(define x '(a b c))
x
=> (a b c)
(car x)
=> a
x
=> (a b c)
(cdr x)
=> (b c)
x
=> (a b c)
```

this is an aspect of functional languages called "referential transparency". The value returned by a function is always the same, given the same inputs. Side effects are not possible because there are no variables.

We can represent a list using box-and-arrow notation.



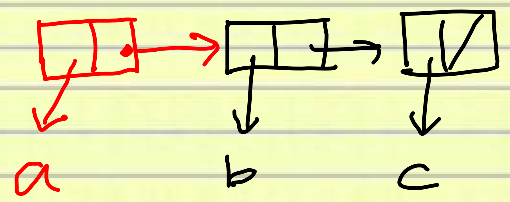
is the list (a (b) c)

Next operation : cons (construct)

cons builds new lists

(cons 'a '(b c)) => (a b c)

(cons '(1) '(2 3)) => ((1) 2 3)



The old list remains unchanged :

(define x '(b c))

(cons 'a '(b c))

=> (a b c)

x

=> (b c)

Last operation: null? — tests whether its argument is the empty list.

```
(null? '()) => #t
(null? '(a b c)) => #f
(null? 1) => #f
(null? 'a) => #f
```

We can drop the quote on the empty list:

```
(null? ()) => #t
```

Simple functions for list processing (LISP)

1. length

e.g. (length '(a b c)) => 3

(length ()) => ∅

(length '((a b) c)) => 2

```
(define length
    (lambda (L)
        (if (null? L)
            ∅
            (+ 1 (length (cdr L))))))
```

Sample derivation:

```
   (length '(a b c))
 = (if (null? '(a b c))
       0
       (+ 1 (length (cdr '(a b c)))))
 = (+ 1 (length '(b c)))
 = (+ 1 (+ 1 (length '(c))))
 = (+ 1 (+ 1 (+ 1 (length ()))))
 = (+ 1 (+ 1 (+ 1 0)))
 = 3
```

The additions are suspended until the recursion "unwinds".

2. member? – treats the list as a set

e.g. (member? 'b '(a b c)) => #t

(member? 'b '(1 2 3)) => #f

We need an operation to test for equality of atoms.

eq?

e.g. (eq? 'a 'a) => #t

(eq? 'a 3) => #f

(eq? '(a b) '(a b)) => #f

```
(define member
    (lambda (x L)
        (if (null? L)
            #f
            (if (eq? x (car L))
                #t
                (member? x (cdr L)))))))
```

```
(member? 'b '(a b c))
= (member? 'b '(b c))
= #t
```

Note that any further occurrences of b will not be found.

This function is tail-recursive because the last call to any function is to the function itself — recursion. Recursion means overhead for each function call. Tail-recursive functions can be optimized by replacing the recursive call with a loop. This is more efficient — we avoid the overhead of multiple calls.

We can rewrite (most) recursive functions with a tail-recursive version and thus get efficient operation.

We will use a facility of Scheme to declare local functions.

[We can write Scheme functions in 2 ways:

```
(define f (lambda (x y) ...))
alternatively (define (f x y) ...)
```
]

Let's rewrite length in a tail-recursive version.

```
(define (length L)
    (define (length1 L n)
        (if (null? L)
            n
            (length1 (cdr L) (+ n 1))))
    (length1 L 0))
```

e.g.
```
   (length '(a b c))
 = (length1 '(a b c) 0)
 = (length1 '(b c) (+ 0 1))
 = (length1 '(c) 2)
 = (length1 () 3)
 = 3
```

The internal loop looks like this:

```
while true do
    (if (null? L) (return n))
    L := (cdr L)
    n := (+ n 1)
end;
```

There is an alternative way to write complex conditionals without nested if-then-else forms.

It's called <u>cond</u> (conditional)

```
(cond   (-test1-   -expr1-)  }
        (-test2-   -expr2-)  } clauses
        (-test3-   -expr3-)  }

        (else   -exprn-)
)
```

Let's rewrite member? using cond :

```
(define (member? x L)
    (cond ((null? L) #f)
          ((eq? x (car L)) #t)
          (else (member? x (cdr L)))))
```

3. Appending of two lists

e.g. (append '(a b) '(c d)) => (a b c d)

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

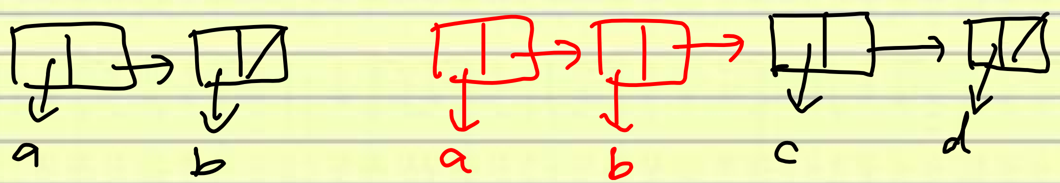
(append '(a b) '(c d))

= (cons 'a (append '(b) '(c d)))

= (cons 'a (cons 'b (append () '(c d))))

= (cons 'a (cons 'b '(c d)))

= (a b c d)



So cons is a copy operation – we copy the first list and append it to the second list.