SEMINARIO LISP

LISP

→ Subconjunto: programación funcional

funciones univaluadas (una única salida)

si necesitamos devolver varios valores → una stru

intérprete de LISP

```
>> (<referencia a función> <argumentos>)
```

EXPRESIÓN LISP A EVALUAR

secuencia de argumentos

Ejemplo:

```
(defun factorial (n)
    (if (= n 0)
        1
      (* n (factorial(- n 1)))))
```

'(2 3 4 5) ≡ (quote (2 3 4 5)) [identidad]

(quote (2 3 4 5)) ↓ evaluación (2 3 4 5)

quote ≡ citar

```
>> (setf lst '(2 4 6))
>> (null lst)
     NIL → '()
         → false
>> (not (null lst))
     true
```

```
>> (cdr '(2 4 6 8))
   (4 6 8)
>> (rest '(2 4 6 8))
   (4 6 8)
>> (car '(2 4 6 8))
   2
>> (first '(2 4 6 8))
   2
```

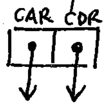
PROHIBIDO USAR EVAL

```
>> (setf duplica #'(lambda (x) (* 2 x)))
>> (apply duplica '(3))
      6
>> (funcall duplica 3)
      6

>> (maucar   #'>   '(2  4  6  8)
                    (1  5  3  2  15))
                     ↓  ↓  ↓  ↓
                     T NIL T  T
```

LISP CRASH COURSE

DEFINITIONS:

- CONS: A cons is a pair of pointers, the first is called the CAR and the second one the CDR CONS 

- ATOM: Basic lisp entity, everything that is not a cons. Such as the empty list, a symbol, a number, a vector, an array, a string...

- LIST: An ordered collection of atoms or lists (the elements of the list) A list is either nil or a cons.

- EXPRESSION: An atom or a list
- FORM: An expression to be evaluated by the Lisp interpreter.
- EVALUATION: → if the form is an atom: the value of the atom
  → if the form is a list: the value of a function evaluatio

- PROPER LIST: A list entity susceptible of being constructed with the list comman
  A proper list is a list that is either nil or a cons whose cdr is a proper list.

- ASSOC-LIST (aka ALIST): A list of conses. Each of those conses represents an association of a given Key with a given value.
  - the CAR of each cons is the Key
  - the CDR of each cons is the value associated with that Key.

Warning! assoc-lists are slow (linear-time access).

COMMANDS FOR LISTS

• CONSTRUCTING LISTS

- CONS: (cons o1 o2) ⇒ (o1. o2)
  Creates a cons, the car is o1 and the cdr is o2.

- LIST: (list o1 ... on) ⇒ (o1 ... on)
  creates a list containing the supplied objects.

- APPEND: (append pl1 ... pln) ⇒ (pl1 ... pln)
  returns a new list that is the concatenation of the given proper lists.
  (append '(a b c) '(d e f) '() '(g)) ⇒ (A B C D E F G)
  (append '(a b c) 'd) ⇒ (A B C . D) → the last one can be an object.
  (append '(a b c) '(d)) ⇒ (A B C D)

- COPY-LIST: (copy-list l) ⇒ l → copy
  returns a copy of the given list. A copy means it is allocated in a different part of the memory.
  (setf lst (copy-list '(1 2 3))) ⇒ (1 2 3)
  ↓
  only copies cdr of the elements

- COPY - TREE: (copy-tree t1) ⇒ t1 → copy
  returns a copy of the given tree
  ✱ check more info

- NCONC: (nconc l1 ... ln) ⇒ (l1 ... ln)
  returns a list that is the concatenation of lists
  ✱ check desktop for more info
  ↓
  destructive

• LIST PROPERTIES

- NULL: (null '(1 2 3)) ⇒ NIL
  (null '()) ⇒ T
  Boolean

- LISTP: (listp '(1 2 3)) ⇒ T
  (listp (cons 1 2)) ⇒ T
  (listp nil) ⇒ T
  (listp (make-array 6)) ⇒ NIL
  Boolean

- LISTS AS SETS

  - MEMBER / MEMBER-IF / MEMBER-IF-NOT: (member 3 '(1 3 5)) ⇒ (

    * check for more info

  - SUBSETP: (subsetp '(3) '(1 3 5)) ⇒ T

    (subsetp '(1 5) '(1 3 5)) ⇒ T

    (subsetp nil '(1 3 5)) ⇒ T

    (subsetp '(4) '(1 3 5)) ⇒ NIL

  - ADJOIN: (adjoin 2 '(1 3 5)) ⇒ (2 1 3 5)

    (adjoin 3 '(1 3 5)) ⇒ (1 3 5)

    Tests whether an item is in the list ↗ it is: nothing chang
    ↘ it isn't: it is inserte

  - UNION: Returns a list that contains every element that occurs in either list1 or list2

    (union '(1 3 5) '(2 4 6)) ⇒ (1 2 3 4 5 6)

  - INTERSECTION: (intersection '(1 3 5 7) '(3 4 6 7)) ⇒ (3 7)

  - SET-DIFFERENCE: Returns a list of elements of list1 that don't appear in list2.

LISTS AS SEQUENCES (sequences = vectors + lists)

- LENGTH: (length seq)

  Returns the number of elements of a sequence

- COUNT: * check for more info

- MAX: (max $n_1 \cdots n_n$)

  Returns the real that is greatest

- MIN: (min $n_1 \cdots n_n$)

  Returns the real that is least.

- FIND / FIND-IF * check for more info

- POSITION / POSITION-IF : * check for more info

- LISTS AS CONSES

  - CAR / FIRST: (car lst) ⇒ "lst[0]"
    returns the first element of a list
  - CDR / REST: (cdr lst) ⇒ "todo menos lst[0]"
    returns the list but the first element.
  - CADR | CAADR / CDAR:

    (caar x) ≡ (car (car x))
    (cadr x) ≡ (car (cdr x))
    (cdr x) ≡ (cdr (car x))

  - NTHCDR | NTHCAR:

    (nthcdr 0 '(1 2 3)) ⇒ (1 2 3)
    (nthcdr 1 '(1 2 3)) ⇒ (2 3)
    (nthcdr 2 '(1 2 3)) ⇒ (3)
    (nthcdr 0 '()) ⇒ NIL ⇐ (nthcdr 3 '())

  - SECOND / THIRD / ... / TENTH: (setf lst '(1 2 3 4 5 6 7 8))

    (second lst) ⇒ 2 ; (fourth lst) ⇒ 4
    (sixth lst) ⇒ 6 ; (ninth lst) ⇒ nil (9 si lo hubiera

  - NTH: (setf lst '(1 2 3 4 5))

    (nth 0 lst) ⇒ 1
    (nth 4 lst) ⇒ 5

    It's like lst[i]

  - LAST: (last x) ⇒ " x[length(x) - 1]"

    pseudocode

- LISTS AS STACKS

  - PUSH: (push element place)
    inserts element in the beginning of place, returning modified pl
  - POP: (pop place)
    extracts the first element of place.

  destructive

- MERGE: * check for more info
- REMOVE: * check for more info
- DELETE: Same as remove but destructive!
- SUBSEQ: (subseq '(1 3 5 7 9 11 13) 3 5) => (7 9)
  Like python array[3:5]
- REVERSE / /NREVERSE destructive!: Returns a new sequence of the same given sequence but in reverse order.

- SORT: * check for more info
  ↓
  destructive!

- EVERY / SOME: * check for more info

- <u>ASSOCIATION LISTS</u>

  - ASSOC: Returns the first cons in the given list whose CAR satisfies the given test, or nil.
    * check for more info

<u>HIGH ORDER FUNCTIONS</u>

- #'<function-name> : reference to a function

- APPLY:

  Arguments → a function
  → a collection of arguments <u>the last of which is a list</u>.

  Evaluates to: value of the function applied to the arguments.

- FUNCALL:

  Arguments → a function
  → a collection of arguments

  Evaluates to: value of the function applied to the arguments.

MAPCAR:

Arguments → a function
→ one or more lists

Evaluates to: list of values resulting from applying the function to each of the elements of the list(s), until some

- MAPCAN :   * check for more info

- MAPLIST :

  Arguments → a function
  → one or more lists

  Evaluates to: list of values resulting from applying the function to the list(s) and to each of the CDRs of the list(s), until some list is exhausted.