FUNCTIONAL ORIENTED PARADIGM

- Functional (***declarative***) paradigm language, i.e., declares a set of functions (sub-problems) that *abstractly* describes the problem to be programmed (*what*) rather than the solution steps (*how*) as in the imperative domain! (functions = abstraction)

Striking features of the "Pure" Functional Paradigm:

1) ***Recursion*** replaces iteration.

2) A ***program is a function*** that, ***mathematically***, ***maps*** inputs to outputs.

3) Application of ***pure*** functions is the central idea (hence ***applicative*** too), they do not alter their parameters.
There is no intermediate results to be kept around, i.e., *internal state*, hence the concept of "memory" is not part of the execution semantics. Hence, there is **no *"side effect"* on the memory's *state***, except at the end of program execution; and ***no "assignment statements"***, but all (S)-expressions.

4) Functions are ***"first-class-citizens"*** passing and returning them to/from other functions (***Polymorphism***).

5) Emphases on ***"function composition"***, rather than statement composition as in the imperative domain.

6) **No globally defined names**, all locally/statically allocated.

7) **No pointer** explicit declarations/assignments.

Examples: **Miranda**, **Haskell**, and **Sisal**

* Unfortunately there was no way to practically and efficiently keep the conceptual view of the "pure" functional paradigm, hence many violations of the "pure" concepts have been implemented as "***impure***" functional languages, the most famous are: LISP, Scheme, and ML.

LISP (LISt Processing)

(John McCarthy, MIT,1961)

- Lisp is functional language widely used in applications like: Artificial Intelligence, Expert Systems, Machine Learning, and Speech Modeling, Auto Theorem Proving, Natural Languages Processing, etc. (why? Easier and more efficient to manipulate "***symbolic data***"-- tabulated data with each cell contains an observation, tabulated prosperities about objects).
- Lisp has the ***syntactic*** equivalence of programs and data (programs are list and lists are programs).
- Symbolic-expressions (Cambridge Polish notation) versus Meta-expressions.
- **Interpreted**, with the possibility of compiled functions, to be invoked at run time; Also, two versions of ***scoping***: ***static*** and ***dynamic***!
- "***Reference count***" dynamic ***garbage collection***.
- "***In-line***" construction of ***aggregate structures.***
- S-expressions: (f (g a b) (h c d))

  M-Expressions: f (g (a, b) , h (c,d))

  Prefix: (+ (* 2 3) (/ 8 2))

  Infix: (2 * 3) + (8/2)
- S-expressions: 1) lists 2) atoms (literal/numeric)
- The first item in a list (S-expression) is considered to be an "operation" with the rest of the list items as its "operands".
- The system will attempt to evaluate any list as an S-expression unless it is "quoted" with ' .

```
% (set 'text '(to be or not to be))
    (to be or not to be)
% text
    (to be or not to be)
```

- Primary data **types** (**no explicit declarations**) are:
  1) ***atoms*** (indivisible S-expression)—
     i) numeric: numbers (integer, real, rational, …)
     ii) literals: symbols

  2) ***lists*** (S-expressions): Consists of atoms and/or lists.
     They are the primary data structure constructors.
     They construct programs (functions) and data.

```
%(make-table   text  nil)     →  make-table (text,nil)
  !----------(code)--------!

%(set 'X '(make-table   text  nil))
            !-----------(data)----------!
```

- **There are also built-in atoms' property lists, with automatic storage management, to hold useful info about each atom.**
- The empty list is considered to be the atom "nil":

```
%(atom ())          %(listp ())
→ t                 → t
```

Important commands:

- When you have an error in your list code you are sent to the debugger:
  0] abort ; gets you out of the debugger to the lisp interpreter level
- In order to exit the lisp interpreter, back to the Linux level:
  *(quit)

Pseudo Functions (procedures):

- They are function with side effect (***impure***) on the memory state.

```
i) set :  %(set 'text '(to be or not to be))    ; bind the second operand to
            (to be or not to be)                ; the first operand.
          % text                          ; text is bound to the above list
            (to be or not to be)
```

ii) ***defun***: Function definition

%(defun f (p_1 p_2 … p_n) body-code)
; the equivalent M-expression:
procedure f (p_1 p_2 … p_n); **begin** <body-code> **end**;

Operations on Lists:

- Lists are ADTs with the following two major sets of operations:
  A) Selectors B) Constructors

A) Selectors Operations ("*pure*" functions):

They are **pure** function without any side effect on their input list (i.e., no change of their input).

i) "**car**": returns the first "element" (atom or list) of it input list:

%(car '(to be or not)) → to
%(car '((to be) or not)) → (to be)

Its input must be a list, yet it might return an atom or a list.

ii) "**cdr**": Returns a copy of its **input list** without its first element:

%(cdr '(to be or not)) → (be or not)
%(cdr '((to be) (or not))) → ((or not))

Its input must be a list, **and it always returns a list**.

They can be combined in a short form for ease of programming:

%(set 'DS '((Don Smith) 45 3000 (August 25 1980)))
→ ((Don Smith) 45 3000 (August 25 1980))
%(car (cdr (cdr (cdr DS)))) or %(cadddr DS)
Both return → (August 25 1980)

%(caddr DS) → 3000
%(caadddr DS) → August
%(cdadddr DS) → (25 1980)

Unfortunately our TCC "Common Lisp" version does not allow more than 4 compressed car&cdr in a sequence! Hence, the last two examples above will not work in our TCC system!

B) **Constructors Operations ("pure" functions):**

i) "cons": (cons <item (atom/list)> <list>)

→ returns a list with the first parameter (list/atom) as its first element and the rest of the elements are the exact element(s) of the second parameter list.

%(cons 'to '(be or not))

→ (to be or not)

%(cons '(to be) '(or not))

→ ((to be) or not)

ii) "append":

(append <list1> <list2>)
→ returns a list of the concatenation of <list1> and <list2>.
(append <list1> <atom>)
→ returns a list of the appending of the atom at the end of <list1>.

%(append '(a b) '(c d)) → (a b c d)

%(append '(a b) 'c) → (a b . c) ; the '.' Is laces before the c to indicate it ; was an atom at the time of the append!

As a good programming habit, list the atom to be appended :

%(append '(a b) '(c)) → (a b c) ;; you get no '.'

OR

%(setq X 'd)
%(append '(a b c) (list X)) → (a b c d)

Definition of the built-in function ***append()***:

```
%(defun append (L M) (cond  ( (null L)                    M
                              )
                            (    t      (cons (car L) (append (cdr L) M) ) )
                       )
  )
  → append              ; takes M and L are two lists
```

There is a more general "append" in our system COMMON LISP where it takes multiple lists, concatenating all of their elements; if the last item is an atom it appends it at the and after placing '.' before it.

%(**append** '(a b) '(c (d e)) '(f g) '(h i))
→ (a b c (d e) f g h i)
%(**append** '(a b) '(c (d e)) '(f g) '(h i) 'j)
→ (a b c (d e) f g h i . j)

iii) "list": takes a number of lists and/or atoms and returns them in a list.

%(list 'a '(b c) 'd '(e (f g)) 'h)
→ (a (b c) d (e (f g)) h)

Information can be Presented Via "**Property Lists**" (*P-list*):

Let P_i be the i^{th} property indicator for the corresponding property value V_i :

$(P_1\ V_1\ P_2\ V_2\ \ldots\ P_n\ V_n)$ is a property list

Example P-list:

(name (Don Smith) age 45 Salary 30000 Hire-Date (August 25 1980))

%(setq Employee-Record '(name (Don Smith) age 45 Salary 30000 Hire-Date (August 25 1980)))

Operations on P-list (user defined):

i) "**getprop**":

```
(defun getprop (P-L  P-N)
 (cond (      (null  P-L)              'undefined-prop          )
       ( (equal (car P-L) P-N)         (cadr P-L)              )
       (   t          (getprop (cddr  P-L)  P-N)  )

  )
 )
→ getprop
```

(getprop Employee-Record 'Hire-Date) ; we did not quote Employee-;Record in order for the system to ;evaluate it!

→ (August 25 1980)

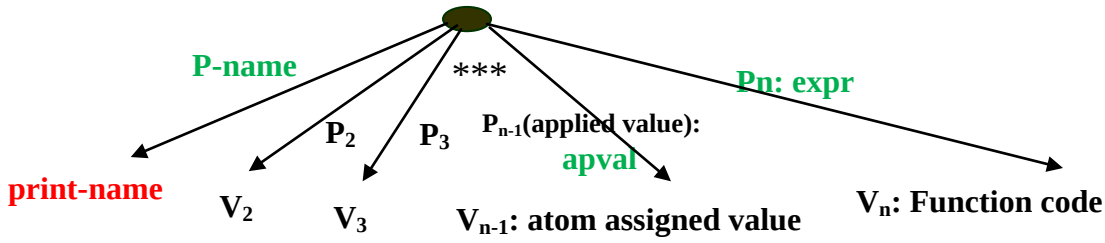
ii) **“putprop”:**

```
%(defun addprop ( P-L  P)
      (setq  P-L (append P-L  P))
)
→ addprop

%(addprop Employee-Record  ‘(address (POBOX 2373 CITY)))

→ (name (Don Smith) age 45 Salary 30000 Hire-Date (August 25 1980) address
     (POBOX 2373 CITY))
```

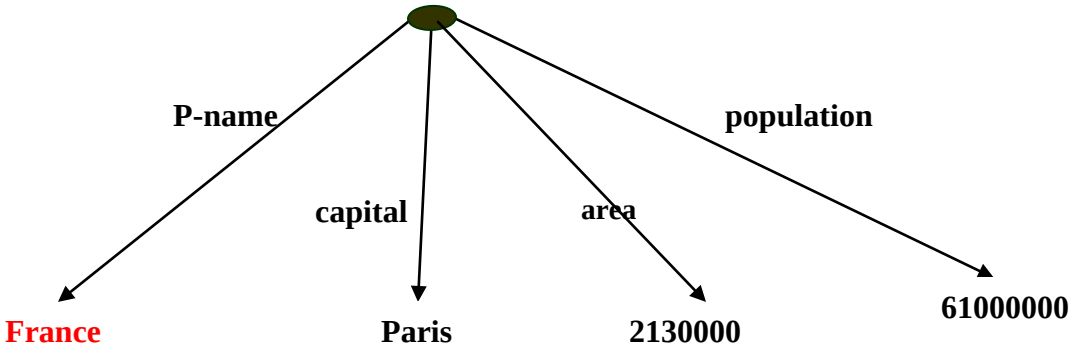
Atoms Have Properties:

- Lisp is developed for AI (Artificial Intelligence) applications (objects with properties). Hence, objects are represented as atoms, each of which has a *p-list*.

“Atom”
with n properties *p-list* [(P_i , V_i) | i:= 1, 2, … n]



There are three built-in properties: ***P-name*** (object name), ***apval*** (applied value), and ***expr*** (expression of the function, **in case of function name to hold its code**, see top of pge 334). The user can add properties via "putprop".



%(putprop 'France 'Paris 'capital)
%(putprop 'France 2130000 'area)
%(putprop 'France 61000000 'population)

In order to recall any property value, simply use the "get" function:

%(get 'France 'capital)
→ Paris

In order to assign a value to an atom, we assign its built in property "apval" such value:

%(set 'Europe '(England France Spain … Poland … Italy))

Or equivalently:

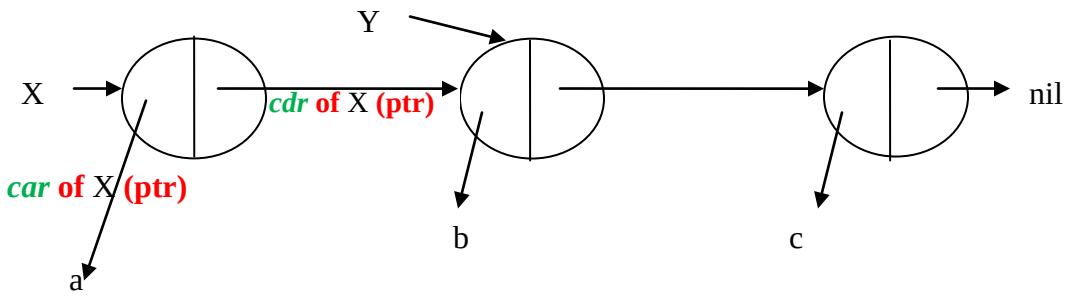
%(putprop 'Europe '(England France Spain … Poland … Italy) 'apval)

%(get 'Europe 'apval) ; to get the value of the atom "Europe"

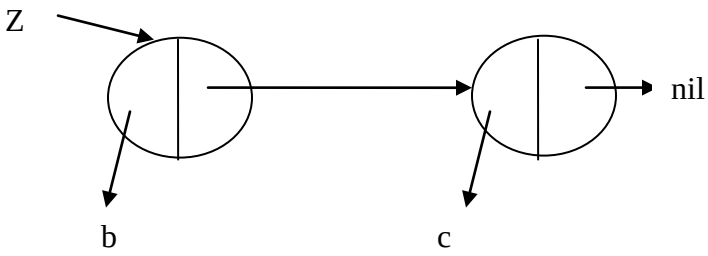
→ (England France Spain … Poland … Italy)

Aliasing in Lisp:

%(setq X ‘(a b c)) %(setq Y (cdr X))



%(equal Y (cdr x)) ; compare structures
→ t
%(eq Y (cdr x)) ; compare pointers
→ t
%(setq Z ‘(b c)) ;



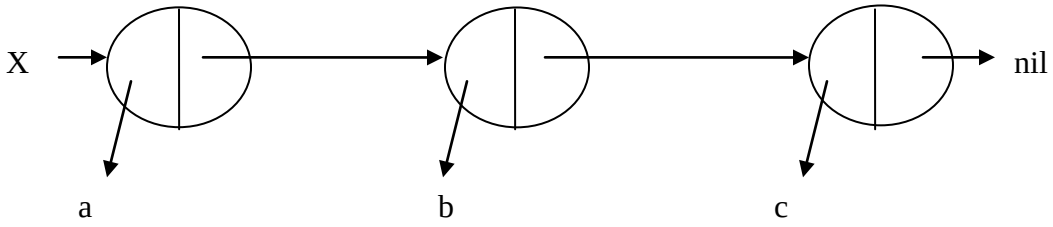
%(equal Y Z)
→ t
%(eq Y Z)
→ nil

Destructive (***impure***) Functions in Lisp, ***rplaca*** and ***rplacd*:-**

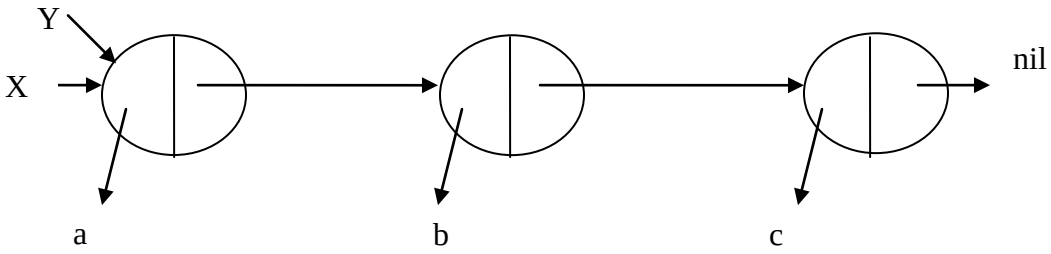
(They alter their inputs!)

i) ***"rplaca":*** It takes two arguments and replaces the car side of its first argument with (left pointer) with its second argument.

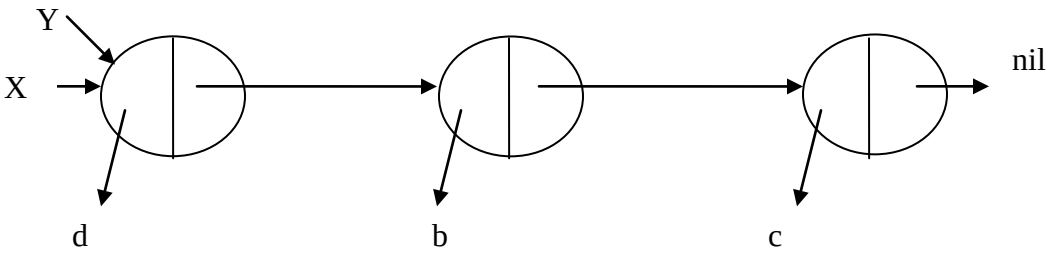
%(setq X '(a b c)) → (a b c)



%(setq Y X) → (a b c) % Y → (a b c)



%(rplaca X 'd) → (d b c)



% Y → (d b c)

Notice that the value of Y changed *implicitly* due to the change in X. Such side effect is due to the *aliasing* of X and Y into same list and the application of *rplaca* (*impure*) function on X. (*security loophole*)

ii) ***"rplacd"***: It takes two arguments and replaces the cdr side of its first argument with (right pointer) with its second argument.

Assume %(setq X '(a b c)) → (a b c)
%(rplacd X 'd) → (a d)

Question: When do you think aliasing would be dangerous in Lisp?
When we start using destructive (impure) functions that alter their inputs.

Mapping Functions:

"mapcar": maps an input function (with one or two parameters) to a list of items, one a time, forming a list of results).

%(mapcar '+ '(1 2 3) '(4 5 6)) ;; Com. Lisp
→ (5 7 9)
%(mapcar 'evenp '(1 2 3 4 7 8 10))
→ (NIL T NIL T NIL T T) ;; Com. Lisp

Functions as "First-Class-Citizens":

* Functions are passed as parameters and also returned as values to/from other functions, respectively.

* In order to be able to return functions from other functions, we need to discuss the "lambda" functions.

*Lambda Function Definition:

A "nameless" function to be used only once in place (and never called again!).

%(mapcar #'(**lambda** (x) (* x x x)) '(10 20 30)) ;; Com. Lisp
→ (1000 8000 27000)

%(**defun** bu (f x) (**function** (**lambda** (y) (**funcall** f x y))))
;; all bold faced words are reserved words in Com. Lisp
→ bu
% **(mapcar (bu '* 5) '(1 2 3))** ; the evaluation of the call
;; (bu '* 5) above returns a one-parameter "lambda" function;;
$*_5$ (multiply-by-5 function) ;;that is mapped on the elements of
;;the input list, one at a time.

→ (5 10 15) ;; the result in Com. Lisp

Name Binding in Lisp:

Done via: 1) property lists (set, putprop, assoc)
2) actual-formal matching
3) "let": environment creator:
*(let ((n_1 e_1) (n_2 e_2) … (n_m e_m)) <code>)
(n_i:name$_i$ e_i:expression$_i$)
The "**let**" provide lexical (static) scoping.

CMU Common Lisp 20a Fedora release 1.fc13 (20A Unicode):

```
GNU CLISP 2.49+ (2010-07-17)
*(setq regular 5)
>> 5
* (defun check-regular () regular)
CHECK-REGULAR
* (check-regular)
>> 5
* (let ((regular 6)) (check-regular))
    ; set regular = 6 at the caller "let"; then call check-regular
>> 6 ;; you get 6 the execution of the body of the "let"
  ; the non-local "regular" inside the code of check-regular is
  ; interpreted in the caller function "let" not at the definer level
  ; (main system level). Hence, the above version of Lisp is
  ; dynamically scoped.
-----------------------------------------------------------------------
Our TCC version of Lisp is GNU CLISP 2.49+ (2010-07-17) and is
statically scoped, i.e., the output will be 5 instead of 6!
```

Some Lisp versions use dynamic and others use static scoping:

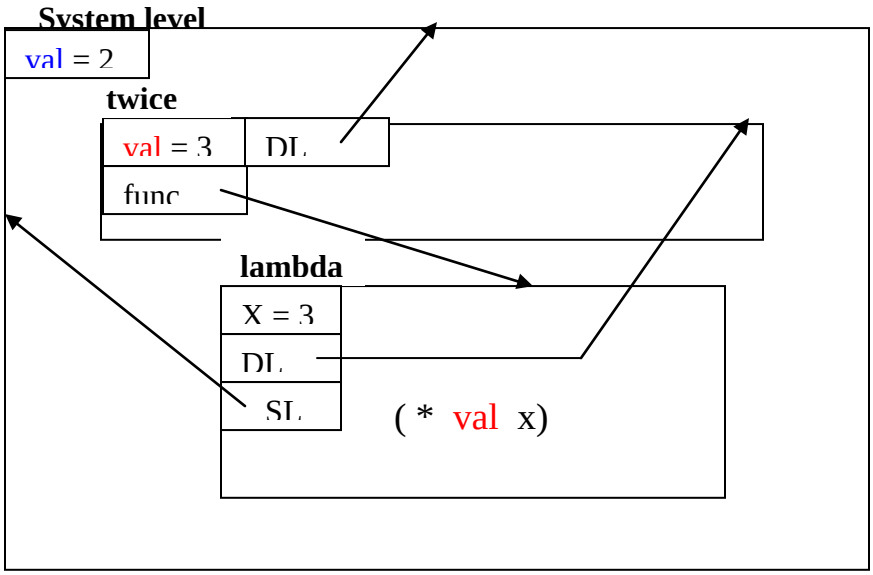
1) **Dynamic Scoping**: Franz lisp
2) **Static Scoping:** Scheme and Common lisp (our system in TCC)

```
%(defun twice (func val)
        (funcall func (funcall  func val)))
    → twice
%(setq val 2)
    → 2
```

%(twice #'(**lambda** (x) (* val x)) 3)

→ 12 ;; static scoping

→ 27 ;; dynamic scopin



The dynamic scoping answer is 27, whereas the static would be 12. The reason is when we encountered the name "val" inside the body of the lambda function we interpreted it in the environment of its caller, i.e., inside "twice" where "val" is a parameter and =3, not in its definer the test code at the system level where val=2.

[verify the **scoping** of our **CMU Common Lisp 20a Fedora release 1.fc13 (20A Unicode**), set the environment of the ***twice()***an run it, as shown above, and check the results 12 vs. 27]

Predicates:

Valid in common lisp:

*(**atom** ‘a)
T

*(**atom** ‘(a b c))
NIL

*(**atom** ())
T

*(**atom** nil)
T

*(**listp** ‘a)
NIL

*(**listp** ‘(a b c))
T

*(**listp** ())
T

*(**listp** ())
T

*(**zerop , oddp, evenp,** < , > , <= , >= , =)

***(member** ‘b ‘(x f g b h y))
(b h y)

***(member** ‘b ‘(x f g (b h y)))
NIL

***(setq** L ‘(x f g b h y))
(x f g b h y)

***(member** ‘b ‘(x f g b h y))
(b h y)

*(**listp** L)
T

We can define our own predicates:

```
*(defun even-between-50-and-100 (x)
  (and (evenp x) (> x 49) (< x 101)))
```

*(**even-between-50-and-100 23)**
NIL

*(**even-between-50-and-100 73)**
T

Some of the standard arithmetic functions are all available:(http://www.n-a-n-o.com/lisp/cmucl-tutorials/LISP-tutorial-4.html)

+, -, *, /, floor, ceiling, mod, sin, cos, tan, sqrt, exp, expt

```
* (+ 3 3/4)            ;type contagion
15/4

* (exp 1)              ;e
2.7182817

* (exp 3)              ;e*e*e
20.085537

* (expt 3 4.2)         ;exponent with a base other than e

100.90418

* (+ 5 6 7 (* 8 9 10))  ;the fns +-*/ all accept multiple arguments
```

Evaluation of LISP (Com.):

1) Very **powerful** language due to:
   i) "recursion",
   ii) the implicit encoding (programming) of atoms very complex hierarchical proprieties (also prop&association lists),
   iii) equivalence of data and programs (both are lists), and
   iv) "lambda" definition of on-line functions (Functions as FCC).
2) **Insecure** aliasing (why? Because of impurity of LISP).
3) **Inefficient** pointer semantics, recursion.
4) Implicit (automatic) garbage collection (pros&cons).
5) **Not "pure"** Functional language [e.g., setq, loop, rplaca, rplacd], taking away the **inherent** ability for exploiting **concurrency** in algorithmic solutions. Independent functions are to be executed simultaneously. (Still there is some way for **lazy** parameters evaluation)

;; A recursive function DFEE which takes a list *L* and an integer *I*
;;and replaces every **even** number in *L* with "2**I*".

```
(defun   DFEE (L  I) (cond  (      (null L)                    nil                              )
                            ( ( evenp (car  L))   (cons   (* 2 I)  (DFEE  (cdr  L)  I  ))     )
                            (         t             (cons     (car L)  (DFEE (cdr  L)  I  ))    )
                      )
 )
```

```
(DFEE   '(11   40   88   15    40    2    100    40    20)    40)
```

→ (11 80 80 15 80 80 80 80 80)

;;; A recursive function add_Lists which takes two
;;; **equal size integer lists** *L1* and *L2* and return their addition
;;; as a list of integers.

```
(defun add_Lists (L1  L2)

  (cond
    ((OR (null L1) (null L2))                    nil                                                  )
    (        t                    (cons (+ (car L1) (car L2)) (add_Lists (cdr L1) (cdr L2))) )
  )
)
```

%(add_Lists '(1 2 3) '(4 5 6))
--> (5 7 9)