SEMINAR 6 – MAP functions in Lisp

"She remembered, as every sensible person does, that you should never shut yourself up in a wardrobe." – The Chronicle of Narnia

MAP functions are functions which apply a given function repeatedly to the elements of the parameter/parameters. In LISP there are several MAP functions, today we will talk about MAPCAR.
Assume that we implement a function, called triple, which takes as parameter a number and returns it multiplied by 3.

```
(DEFUN triple (x) (* x 3))
```

Using ***MAPCAR*** we can apply this function to all the elements of a list. For example, (MAPCAR #'triple '(1 2 3 4 5)) is equivalent to calling *triple* on each element of the list (so *(triple 1) (triple 2) (triple 3)*…). Each individual call returns a value, and MAPCAR gathers them in a list, using the function *list.* So:

(mapcar #'triple '(1 2 3 4 5))
is equivalent with
(list (triple 1) (triple 2) (triple 3) (triple 4) (triple 5))
and the result will be the list (3 6 9 12 15)

What if the list contains non-numerical atoms? For example if we call (MAPCAR #'triple (1 a 2 b 3 c))? It will give an error, since triple will have an error, due to the fact that the parameter is not a number: *argument to * should be a number: A*. As a solution we can modify the triple function to treat the case when the atom is non-numeric:

```
(DEFUN triple (x)
  (COND
       ((numberp x) (* x 3))
       (t x)
  )
)
```

Now the call (MAPCAR #'triple '(1 a 2 b 3 c)) will return the list (3 A 6 B 9 C).
What if the list is non-linear and contains sublists? For example if we call (MAPCAR #'triple '(1 a (2 b) 3 c))? We will not have an error, since *triple* will not give an error if the parameter is a list, but enters the true branch and returns the unmodified list. So the result will be (3 A (2 B) 3 C).

How could we triple the elements from the sublists? We can observe that this is exactly what MAPCAR does, applies the function *triple* on every element of the list, so we can change the implementation of function *triple* in the following way:

```
 (DEFUN triple (x)
      (COND
             ((numberp x) (* x 3))
             ((atom x) x)
             (t (mapcar #'triple x))
      )
)
```

And now we do not even need to call the function as (MAPCAR #'triple '(1 a (2 b) 3 c)), we can call directly (triple '(1 a (2 b) 3 c))): since the parameter is a list the execution will enter the last branch and MAPCAR will be called on the elements of the list.

#'fname designates that the argument *fname* is a function name, while ***'sym*** is for symbols; some dialects work without #; it's use often with mapcar and apply.
And the recursive mathematical model for the triple function will be:

$$triple(x) = \begin{cases} 3 * x, if\ x\ is\ a\ number \\ x, if\ x\ is\ an\ atom \\ \bigcup_{i=1}^{n} triple(x_i)\,, if\ x\ is\ a\ list \end{cases}$$

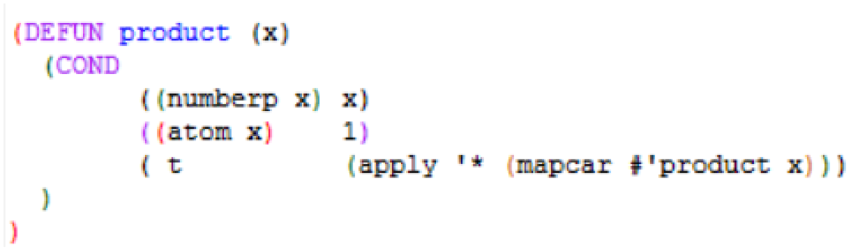
Use APPLY.
MAPCAR returns a list. If we want the result to be a number, we can use the ***apply* function** to apply on the resulting list a function which computes the final result.

(APPLY function set_of_parameters) : val

Example: (apply #'+ (1 2 3 4 5)) => 15
(apply #'max (1 2 3 4 5)) => 5 ; max is lisp function built-in

For example, if we want the product of the numbers from the list:

$$product(x) = \begin{cases} x, \quad if\ x\ is\ a\ number \\ 1, \quad if\ x\ is\ atom\ but\ not\ number \\ \prod_{i=1}^{n} product(x_i), if\ x\ is\ a\ list \end{cases}$$

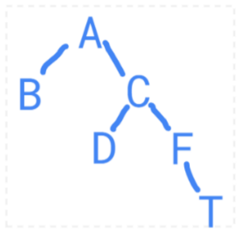


```
(DEFUN product  (x)
  (COND
         ((numberp x)  x)
         ((atom x)     1)
         ( t           (apply '* (mapcar #'product x)))
  )
)
```

```
(product '(1 2 3 (4 ) 5 a b ) )     =>  120
```

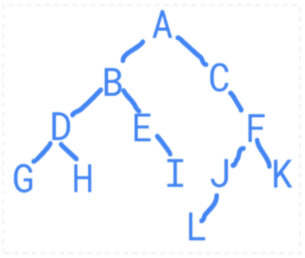
N-ary trees
Tres can be represented as a list of the form (root (subtree1) (subtree2) ... (subtreeN)), where root is the root of the tree with branches several subtrees that ar also represented as a list (rootX (subtreeX1) (subtreeX2) ... (subtreeXn)) .
Example: (A (B (F) (G) (H (I))) (C (E)) (D))

Binary tree
As a list: (root left_subtree right_subtree) Example: (A (B) (C (D) (F (T))))
List of node followed by number of children: (A 2 B 0 C 2 D 0 F 1 T 0)



Problems

1. ***Compute the number of nodes from the even levels of an n-ary tree, represented as (root (subtree_1) (subtree_2) ... (subtree_n)). The level of the root is 1. For example, for the tree (A (B (D (G) (H)) (E (I))) (C (F (J (L)) (K)))) the result is 7.***



MAPCAR helps us to process all levels of the tree, but in order to know which nodes has to be counted and which has not to be counted, we need a parameter to show the current level. If the current level is even and we have found a node (meaning that the parameter is an atom, not a list) we will count the node. If we found a node, but on an odd level, we do not count it, and if we find a subtree (a list), we keep on searching in the subtree using MAPCAR and we increment the current level.

$$countEven(tree, level) = \begin{cases} 1, & tree\ is\ an\ atom\ an\ level\ is\ even \\ 0, & tree\ is\ an\ atom \\ \sum_{i=1}^{n} countEven(tree_i, level+1) & \end{cases}$$

This time we combine MAPCAR with a function with 2 parameters. When there are more than one parameters MAPCAR expects to receive lists and will apply the function on the first elements of the lists, then on the second elements of the lists, and so on, until the shortest list ends. The problem is that parameter 2 in our case is not a list, so if we write *(MAPCAR #'countEven tree (+ 1 nivel))* we will have an error since the last parameter will be treated as a list and MAPCAR will try to take the CAR of the list.
Examples of mapcar with several parameters:

```
(mapcar 'list '(A B C) '(D E F)) = ((AD) (BE) (CF))
(mapcar 'list '(A B C) '(D E)) = ((AD) (BE))
```

The solution in this case is to use a ***lambda expression***. Lambda expressions are in general simple functions, without a name, defined directly where they are going to be used. ***A lambda expression helps us to „transform" our function with 2 parameters into a function with one single parameter***.

When writing a definition of a function we use **defun** which has the form:

```
(defun functionname (parameter1 parameter2 ... ) (body_of_the_function))
```

And we call this function:

```
(functionname  parameter1 parameter2 ...)
```

When using a lambda function it is similar, with the exception that the lambda function does not have a name, and for this reason we use directly the definition of the lambda function instead of its name when calling it:

```
(lambda (parameter1 parameter2 ...) (body_of_the_function))
```

With map functions, in our case it will look like this:

```
( mapcar     #'(lambda (par1) (body))     inputlist )
```

```
(DEFUN countEven(tree level)
  (COND
        ((and (atom tree) (= (mod level 2) 0)) 1)
        ((atom tree)                          0)
        ( t (apply '+ (mapcar #'(lambda (a) (countEven a (+ 1 level))) tree)))
  )
)
```

We need a main function to call *countEven* and to initialize the parameter for the level.

$$nodes(tree) = countEven(tree, 0)$$

```
(DEFUN nodes (tree) (countEven tree 0))
```

An alternative solution for this problem makes use of the particular fact that in each tree/subtree there is only one node (atom) that may qualify the requirement if the level is even or odd and add 1 to the sum. Therefore, at each hierarchical level, the solution adds the parity of the level directly which is 1 or 0 (corresponding to one node) and calls the mapcar and countEven for the the tail of the list (that contains only lists of subtrees) for the next levels. This will count in the end the number of nodes on even levels. (solution provided by Tudor Jinga group 924).

```
(defun countEven2 (tree evenLevel)
    (+ evenLevel
       (apply '+
           (mapcar (lambda (tree) (countEven tree (mod (+ 1 evenLevel) 2))) (cdr tree))
       )
    )
)
```

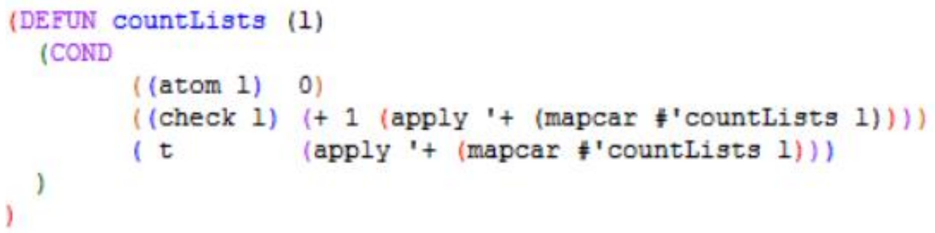
Based on the same idea, the following solution returns 0 at atom level (meaning when it receives an empty list, when the list contains only an atom/node and cdr is empty) and it adds only lists that are on even levels. This time level is the actual number of the level, and because we counting is of lists that are on even levels, and not atom from the even level, we start with level 1 (the list that contains the root is at level 1). (solution proposed by Mihai Grebra in group 923).

```
;treeCE(tree, level)={0  , if tree is atom (includes [])
;                     {1 + sum(treeCE(ti, level+1)) i=2, n  , if tree=t1t2..tn list & level %2=0
;                     { sum(treeCE(ti, level+1), i=2, n , if tree=t1t2..tn & level%2=1


(defun treeCE (tree level)
  (cond
    ((atom tree) 0)
    ((= 0 (mod level 2))  ( + 1 (apply '+ (mapcar #'(lambda (ti) (treeCE ti (+ 1 level))) (cdr tree)))))
    ( t  (apply '+ (mapcar #'(lambda (ti) (treeCE ti (+ 1 level))) (cdr tree))))
  )
)

(defun maintreeCE (tree)
 (treeCE tree 1)
)
```

2. ***You are given a nonlinear list. Compute the number of sublists (including the initial list) where the first numeric atom is 5. For example, for the list: (A 5 (B C D) 2 1 (G (5 H) 7 D) 11 14) there are 3 such lists: the initial list, (G (5 H) 7 D) and (5 H).***

   - We will use two functions for solving the problem: we need a function which, given a list, checks if it respects the condition, meaning that the first numerical atom is 5. The other function will count how many lists fulfill the condition, using the first functions.
   - We'll start with the second function. Assume that the checking function is called ***check*** and it returns T if the first number in the list is 5 and nil otherwise.

$$countLists\ (l) = \begin{cases} 0, if\ l\ is\ atom \\ 1 + \sum_{i=1}^{n} countLists(l_i), if\ l\ is\ a\ list\ and\ check(l)\ is\ true \\ \sum_{i=1}^{n} countLists(l_i), otherwise \end{cases}$$



```
(DEFUN countLists (l)
  (COND
        ((atom l)   0)
        ((check l)  (+ 1 (apply '+ (mapcar #'countLists l))))
        ( t         (apply '+ (mapcar #'countLists l)))
  )
)
```

   - How do we check if the first numerical atom is 5 or not? The elements of the list can be of three types (numerical atom, non-numerical atom and list). And we have to consider the case when we get to the end of the list:
     - Empty list – return nil
     - Numeric atom – check if it is 5 or not
     - Nonnumeric atom – keep on checking the list
     - List – This one is complicated...if the sublist contains at least one numerical atom, then we only care for the sublist. If it contains no numeric atoms, we have to keep checking the rest of the list.

Possible solutions:

- Before checking, linearize the list, so that sublists disappear.
- Write a function to check if a list contains a numeric atom or not.
- Make the check function return 3 different values, not just T and nil.

```
; linearize list with numeric values
```

$$transform(l_1 l_2 .. l_n) = \begin{cases} \emptyset, \quad if\ l = \emptyset \\ l_1\ U\ transform(l_2 \ldots l_n), \quad if\ l_1\ is\ a\ number \\ transform(l_2 .. l_n), if\ l_1\ is\ a\ number \\ append\big(transform(l_1), transform(l_2 .. l_n)\big), otherwise \end{cases}$$

Where ***append*** will return the reunion of two lists (flatten one level).

```
(DEFUN transform(l)
   (COND
      ((null l)            nil)
      ((numberp (car l))  (cons (car l) (transform (cdr l))))
      ((atom (car l))     (transform (cdr l)))
      ( T                 (APPEND (transform (car l)) (transform (cdr l))))
   )
)
```

```
;check if first numeric element is 5
```

$$check(l_1l_2..l_n) = \begin{cases} false, & if\ transform(l_1l_2\dots l_n) = \emptyset \\ true, & if\ transform(l_1l_2\dots l_n) = t_1t_2..t_n\ and\ t_1 = 5 \\ false, & otherwise \end{cases}$$

```
(DEFUN check (l)
  (COND
     ((null (transform l))            nil)
     ((equal (car (transform l)) 5)   T)
     ( T                              nil)
  )
)
```

- Alternatively, let's implement the transform function using map functions (to linearize the list, selecting only numerical elements). Use reunion of lists like append function to flatten the list.

$$transmap(l) = \begin{cases} \emptyset, & if\ l = \emptyset \\ \{l\}\ , & if\ l\ is\ a\ number \\ \emptyset, & if\ l\ is\ atom\ nonnumeric \\ \bigcup_{i=1}^{n} transmap(l_i)\,, & if\ l\ is\ a\ list \end{cases}$$

```
(defun transmap ( l)
  (cond
     ((null l)  nil)
     ((numberp l)  (list l))
     ((atom l)  nil)
     ( t     (apply #'append  (mapcar #'transmap  l)))
  )
)
```

Or, we can use mapcan (which will call function transmap on each element of l, but combine results with NCONC instead of LIST, resulting in concatenating the results similar with the effect of append and flattening the list). So, we don't need to apply append anymore. The last branch becomes:

```
(t     (mapcan #'transmap l) )
```

For example, if the input list l=l1 l2 .. ln, the above would be equivalent with:

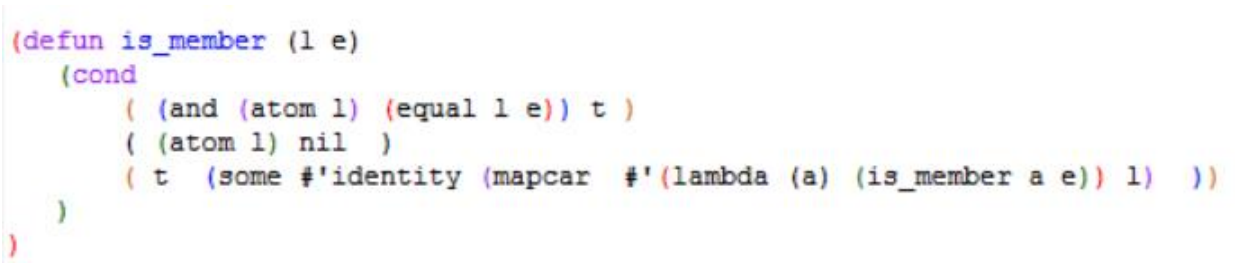
```
(nconc (transmap l1 ) (transmap l2 ) … (transmap ln) )
```

Where l1=(car l),   l2=(cadr l), l3= (caddr l), …

3. *Alternatives to AND/OR (which are not functions but operators – although some arithmetic operators are functions in LISP, for example +, -, * etc…) to use with apply:* *SOME (similar to OR) , EVERY (similar to AND).*
Identity ***is intended for use with functions that require a function as an argument, could be defined as: (defun identity (x) x) .***

a) Write a function that determines if e is an element of a non-linear list (use logical OR).

$$is\_member(l,e) = \begin{cases} t, & if\ l\ atom\ and\ \ l = e \\ nil, & if\ l\ is\ atom\ and\ l \neq e \\ \bigvee_{i=1}^{n} is\_memeber(l_i, e) & \end{cases}$$



```
(defun is_member (l e)
   (cond
        ( (and (atom l) (equal l e)) t )
        ( (atom l) nil   )
        ( t   (some #'identity (mapcar  #'(lambda (a) (is_member a e)) l)   ))
   )
)
```

```
(is_member '(1 2 (8 7) 3 4) 7)    => T
(is_member '(1 2 3 4) 7)          => NIL
```

b) Write a function that determines if all elements in a non-linear list are equal to e (use logical AND).

$$all\_like(l,e) = \begin{cases} t, & if\ l\ is\ atom\ and\ l = e \\ nil, & if\ l\ is\ atom\ and\ l\ \neq e \\ \bigwedge_{i=1}^{n} all\_like(l_i, e) & \end{cases}$$

```
(defun all_like (l e)
   (cond
       ( (and (atom l) (equal l e)) t )
       ( (atom l) nil  )
       ( t  (every #'identity (mapcar  #'(lambda (a) (all_like a e)) l)  ))
   )
)
```

```
(all_like '(2 (2 2) 2) 2)      =>   T
(all_like '(1 2 3) 2)          =>   NIL
```

4. *Other map functions:*

```
(mapcar #'triple '(1 2 3))    => (3 6 9)
(maplist #'triple '(1 2 3))    => ((3 6 9) (6 9) (9))
(mapcan #'triple '(1))        =>   3
(mapcon #'triple '(1 2 3))     => (3 6 9 6 9 9)
```