Seminar 6 –MAP functions

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.
And the recursiv mathematical model for the triple function will be:

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

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

$$product(x) = \begin{cases} x, if\ x\ is\ a\ number \\ 1, if\ x\ is\ atom \\ \prod_{i=1}^{n} product(x_i), if\ x\ is\ a\ list \end{cases}$$

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

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. 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.

```
(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))
```

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.