Seminar 5
Recursive programming in Lisp

Today we start working in Lisp. The problems that we are going to solve will be similar to the problems solved in Prolog, the recursive mathematical models will be almost identical, the difference will be in the actual implementation.

- In Lisp, lists are by default heterogeneous. Thus, unless it is explicitly specified that we have a linear list, we have to assume that we have sublists (which can contain other sublists). Similarly, we assume by default, unless specified otherwise, that the elements of the list are both numerical and nonnumerical atoms.
- Information in Lisp is represented as a list. Not just the data that we will work with is a list, but the code itself which does the processing is a list as well. Every list has as first element the name of the function to apply, and the other elements of the list are parameters of the function. For example:
    - Max is a function which returns the maximum of its parameters (can have any number of parameters, but they have to be numbers). In other programming languages, max would be called as: *max (1,2,3,4,5,6,7)*. In Lisp, the call becomes *(max 1 2 3 4 5 6 7).*
    - If we want the first element of the list to not be considered a function (because for example is a simple list which is our parameter), we have to place an apostrophe in front of the list: '(1 2 3 4 5 6). If we write only the list, without the apostrophe, Lisp will give us an error: *1 is not a function name.*
- When a new function is defined, this function is a list as well, it starts with the *defun* keyword, followed by the name of the function and the parameter list and body of the function.
-
```
(defun functionName (param1 param2 ...) function_body)
```

- The function body in general contains several forms which can be conditions and instructions to be executed if the condition is true (very similar to the mathematical model). For these conditions we use the *cond* function (similar to *swith* from other programming languages) which can contain any number of conditions.

```
(defun functionName (param1 param2 …)
(cond
      (cond_1 ins_1)
      (cond_2 ins_2)
      (cond_3 ins_3)
      …
      (cond_n ins_n)
)
)
```

1. Define a function which merges, without keeping the doubles, two sorted linear lists.

$$merge\ (l_1 l_2 \dots l_n, k_1 k_2 \dots k_m) = \begin{cases} l_1 l_2 \dots l_n, if\ m = 0 \\ k_1 k_2 \dots k_m, if\ n = 0 \\ l_1 \cup merge\ (l_2 \dots l_n, k_1 \dots k_m), if\ l1\ < k_1 \\ k_1 \cup merge\ (l_1 \dots l_n, k_2 \dots k_m), if\ k_1 < l_1 \\ l_1 \cup merge\ (l_2 \dots l_n, k_2 \dots k_n), if\ l_1 = k_1 \end{cases}$$

```
(defun merge (l1 l2)

	(cond
		((null l2) l1)
		((null l1) l2)
		((< (car l1) (car l2)) (cons (car l1) (merge (cdr l1) l2)))
		((> (car l1) (car l2)) (cons (car l2) (merge l1 (cdr l2))))
		(t (cons (car l1) (merge (cdr l1) (cdr l2))))
	)
	)
```

- Lisp has 3 functions which can be used to create lists: *cons*, *list* and *append*. Let's see how the results looks like based on the parameters:

	cons	**list**	**append**
'A 'B	(A . B)	(A B)	Error
'A '(B C D)	(A B C D)	(A (B C D))	Error
'(A B C) '(D E F)	((A B C) D E F)	((A B C) (D E F))	(A B C D E F)
'(A B C) 'D	((A B C) . D)	((A B C) D)	(A B C . D)
'A 'B 'C 'D	Error	(A B C D)	Error
'(A B) '(C D) '(E F)	Error	((A B) (C D) (E F))	(A B C D E F)
'(A B) 'C '(E F) 'D	Error	((A B) C (E F) D)	Error
'(A B) '(E F) 'D	Error	((AB) (E F) D)	(A B E F . D)

2. Define a function to remove all the occurrences of an element from a nonlinear list.

$$removeAll(l_1 \dots l_n, e) = \begin{cases} \emptyset, n = 0 \\ removeAll(l_1) \cup removeAll(l_2 \dots l_n), if\ l_1 is\ a\ list \\ removeAll(l_2 \dots l_n), if\ l_1 = e \\ l_1 \cup removeAll(l_2 \dots l_n), otherwise \end{cases}$$

```
(defun removeAll (l e)
(cond
	((null l) nil)
	((listp (car l)) (cons (removeAll (car l) e) (removeAll (cdr l) e)) )
	((equal (car l) e) (removeAll(cdr l) e))
	(t (cons (car l) (removeAll (cdr l) e)))
)
)
```

3. Build a list with the positions of the minimum number from a linear list.

- One version is to use 3 functions to solve the problem:
  - A function to find the minimum of a linear list (which can contain nonnumerical atoms)
  - It is important to observe that a solution where we keep checking if the first element is the minimum (re-computed at every iteration) is not correct, because the minimum of the list can change as we process it. For example, the list (1 2 3 4 5) will always contain

as first element the minimum (of the current list): 1 is the minimum of (1 2 3 4 5), 2 is the minimum of (2 3 4 5) etc. This is why you either have to transmit the minimum as a parameter (and actually find the positions of a given element in this function) or transmit the copy of the list and compute the minimum for that copy.

- A third function to combine these two.

$$minim(l_1 \dots l_n) = \begin{cases} 10000, n = 0 \\ \min(l_1, minim(l_2 \dots l_n), if\ l_1 is\ a\ number \\ minim(l_2 \dots l_n), otherwise \end{cases}$$

```
(defun minim (l)
(cond
      ((null l) 10000)
      ((numberp (car l)) (min (car l) (minim (cdr l))))
      ((atom (car l)) (minim (cdr l)))
)
)
```

$$positions(l_1 \dots l_n, e, pc) = \begin{cases} \emptyset, n = 0 \\ pc \cup positions(l_2 \dots l_n, e, pc + 1), if\ l_1 = e \\ positions(l_2 \dots l_n, e, pc + 1), otherwise \end{cases}$$

```
(defun positions (l e p)
(cond
      ((null l) nil)
      ((equal (car l) e) (cons p (positions (cdr l) e (+ 1 p))))
      (t (positions (cdr l) e (+ 1 p)))
)
)
```

$$pozMain(l_1 \dots l_n) = positions(l_1 \dots l_n, minim(l_1 \dots l_n), 1)$$

```
(defun pozMain (l) (positions l (minim l) 1))
```

There is another approach for solving the problem with one single traversal in which we both look for the minimum and build the list of positions. We will have a *current minimum* (a collector variable), a current positions and a list with the positions where we have found the *current minimum* (another collector variable). At every step the first element of the list can be:

- A nonnumeric atom – go on, ignore the element, increase the current position
- A numeric atom equal to the current minimum – we have found a new position for our minimum, add this to our list of positions of minimum
- A numeric atom less than the current minimum – our current minimum is not the minimum of the list. Change the current minimum and the list of the positions for the current minimum will be a new list containing only the current positions.
- A numeric atom greater than the current minimum – ignore it, go on.

When the initial list becomes empty in the collector list we have the positions of the minimum.

How do we initialize the current minimum?

- One approach is to put the first element of the list as the current minimum. But this element can be a nonnumeric atom. If we do this we have to change our algorithm described above (which is based on the

idea that the minimum is a number) and add another branch to check if the current element is a numeric atom and the minimum is nonnumeric, simply set this current element as minimum.

- A similar approach is to start the minimum with the value *nil*. We still need the extra branch described above.
- A third approach is to implement a function which finds the first numeric atom of the list and initialize the minimum with this value.

We will implement the case when the *current minimum* is initialized with the first element of the list.

$$pozMin\ (l_1 l_2 \dots l_n, minC, pozC, listPoz) = \begin{cases} listPoz, n = 0 \\ pozMin(l_2 \dots l_n, minC, pozC + 1, listPoz), l_1 is\ nonnumeric\ atom \\ pozMin(l_2 \dots l_n, l_1, pozC + 1, (pozC)), minC\ is\ not\ a\ number \\ pozMin(l_2 \dots l_n, minC, pozC + 1, listPoz\ U\ pozC), minC = l_1 \\ pozMin\big(l_2 \dots l_n, l_1, pozC + 1, (pozC)\big), \quad l_1 < minC \\ pozMin(l_2 \dots l_n, \quad minC, pozC + 1, listPoz), otherwise \end{cases}$$

```
(defun pozMin (l minC pozC listPoz)
(cond
       ((null l) listPoz)
       ((not (numberp (car l))) (pozMin (cdr l) minC (+ pozC 1) listPoz))
       ((not (numberp minC)) (pozMin (cdr l) (car l) (+ pozC 1) (list pozC)))
       ((= minC (car l)) (pozMin (cdr l) minC (+ pozC 1) (append listPoz (list pozC))))
       ((< (car l) minC) (pozMin (cdr l) (car l) (+ pozC 1) (list pozC)))
       (t (pozMin (cdr l) minC (+ pozC 1) listPoz))
)
)
```

We need the main function to perform the first call and initialize the parameters:

$$pozMinMain(l_1 \dots l_n) = pozMin(l_1 \dots l_n, l_1, 1, \emptyset)$$

```
(defun pozMinMain (l)
       (pozMin l (car l) 1 nil)
)
```

4. Given a linear list, add elements from N to N.

$$addN(l_1 \dots l_n, e, pc, N) = \begin{cases} \emptyset, n = 0 \\ e \cup addN\ (l_1 \dots l_n, e, pc + 1, N), if\ pc\ mod\ N = 0 \\ l_1 \cup addN(l_2 \dots l_n, e, pc + 1, N), otherwise \end{cases}$$

```
(defun addN (l e pc n)
(cond
       ((null l) nil)
       ((equal 0 (mod pc n)) (cons e (addN l e (+ 1 pc) n)))
       (t (cons (car l) (addN (cdr l) e (+ 1 pc) n)))
)
)
```

- And the main function to call addN.

$$addNMain(l_1 \dots l_n, e, N) = addN(l_1 \dots l_n, e, 1, N)$$

```
(defun addNMain (l e n)
       (addN l e 1 n)
)
```