LECTURE 9 – FUNCTIONS, EXAMPLES, BINARY TREES

Contents

1. Objects management in Lisp. OBLIST and ALIST

A Lisp object is a structure made up of:

- the name of the symbol;
- its value;
- the list of properties associated with the symbol.

The management of the symbols used in a Lisp program is performed by the system with the help of a special table called the list of objects (OBLIST). Any symbol is a unique object in the system. The introduction of a new symbol determines its addition to the OBLIST. At each occurrence of an atom, it is searched in OBLIST. If it is found, its address is returned.

To implement the calling mechanism, the Lisp system uses another list, called the argument list (ALIST). Each element in ALIST is a dotted pair consisting of a formal parameter and its associated argument or value. In general, a function defined with n parameters P1, P2, ..., Pn and called by (F A1 A2 ... An) will add to ALIST n pairs looking like (Pi . Ai) if the function evaluates its arguments or (Pi . Ai'), where Ai' is the value of the argument Ai, if the Lisp environment evaluates the arguments. For example, for the function

```
(DEFUN F (A B)
    (COND
        ((ATOM B) A)
        (T (CONS A B))
    )
)
```

called with (F 'X '(1 . 1)), we will have for ALIST the structure (... (A . X) (B . (1 . 1))).

The symbols are represented in the structure by their address in the OBLIST, and the numerical atoms and the strings by their value directly in the cell that contains them.

Initially, at the beginning of the program, ALIST is empty. As new functions are evaluated, pairs are added to the ALIST, and when the function evaluation is completed, the pairs created on the call are deleted. In the body of the function being evaluated, the value of a symbol s is first searched in the pairs in ALIST starting from the top to the base (this action is the one that dynamically establishes the field of visibility). If the value is not found in ALIST, the search in OBLIST is continued. Therefore, it is clear that ALIST and OBLIST form the current context of program execution.

Let's look at an explanatory example. Consider the following evaluations:

- (SETF X 1) initializes the symbol X with the value 1;
- (SETF Y 10) initializes the symbol Y with the value 10;
- (DEFUN INC (X) (SETF X (+ 1 X))) defines a function to increase the value of the parameter;
- (INC X) is evaluated at 2;

- X is also evaluated at 1;
- (INC Y) is evaluated at 11;
- Y is also evaluated at 10.

Note, therefore, that the changes made to the values of symbols representing the formal arguments will be lost after leaving the body of the function and returning to the calling context.

2. Functions with destructive access to the contents of the lists

The Lisp workspace image is a web of pointers between system objects. Therefore, it is necessary for the user to know the exact effects of the evaluation process in this huge graph, especially the action of certain functions of modifying structures, as well as understanding aspects related to the choice between sharing and copying, equality of structure. or equality of address, consumption and release of space as well as the cost of accessing data spaces.

The class of functions that we present below allows the modification (destructive) of some existing structures.

RPLACA subr 2 (l e): l
RPLACD subr 2 (l e): l

These functions replace the value of the CAR field (respectively CDR) of l with e, returning the new list as a result.

- (SETQ L '(A B C))
- (RPLACA L 'D) is evaluated at (D B C)
- L is evaluated at (D B C)
- (SETQ F '(CONS A B))   F:= (A B)

- (RPLACA F 'LIST) is evaluated at (LIST A B) ?
- (RPLACD F (CDDR F)) is evaluated at (LIST B)
- (SETQ X '(LISP IS A LANGUAGE))
- (SETQ Y X)
- (RPLACA X 'C) is evaluated at (C IS A LANGUAGE)
- X is evaluated at (C IS A LANGUAGE)
- (RPLACD X '(IS EASY)) is evaluated at (C IS EASY)
- Y is evaluated at (EASY)

On the other hand:

- (SETQ X '(LISP IS A LANGUAGE))
- (SETQ Y X)
- (SETQ X (CONS (CAR X) '(IS EASY))) is evaluated at (LISP IS EASY)
- X is evaluated at (LISP IS EASY)
- Y is evaluated at (LISP IS A LANGUAGE)

NCONC lsubr 1- (f l1 l2 ... ln): l

Regarding the modification or not of the structure of the lists involved, the concatenation of lists can be performed in two ways: with the modification of the lists (using the NCONC function) and without (using the APPEND function)

NCONC performs the actual (physical) concatenation by modifying the last pointer (with the NIL value) of the first n1 arguments and returns the first argument, which will include all the others at the output.

- (SETQ L1 '(A B C) L2 '(D E)) is evaluated at (D E)
- (APPEND L1 L2) is evaluated at (A B C D E)
- L1 is evaluated at (A B C)

- L2 is evaluated at (D E)
- (SETQ L3 (NCONC L1 L2)) is evaluated at (A B C D E)
- L1 is evaluated at (A B C D E)
- (SETQ L1 '(A) L2 '(B) L3 '(C)) is evaluated at (C)
- L1 is evaluated at (A)
- L2 is evaluated at (B)
- L3 is evaluated at (C)
- (NCONC L1 L2 L3) is evaluated at (A B C)
- L1 is evaluated at (A B C)
- L2 is evaluated at (B C)
- L3 is evaluated at (C)

REMOVE subr 2 (el): l

Returns a copy of the list argument in which all EQL elements with e on the surface level have been deleted.

DELETE subr 2 (el): l

It is the destructive version of the REMOVE function, with the observation that the first element will not be destroyed destructively (there is no previous cell to which the CDR field will be modified).

- (SETQ Z (LIST 'A '(B C) 'D 'D)) is evaluated at (A (B C) D D)
- (REMOVE 'A Z) is evaluated at ((B C) D D)
- Z is evaluated at (A (B C) D D)
- (REMOVE 'D Z) is evaluated at (A (B C))
- Z is evaluated at (A (B C) D D)
- (DELETE 'A Z) is evaluated at ((B C) D D)
- Z is evaluated at (A (B C) D D)
- (DELETE 'D Z) is evaluated at (A (B C))

- Z is evaluated at (A (B C))

A possible implementation of the REMOVE function is:

```
(DEFUN REMOVE (e l)
        (COND
                ((ATOM l) l)
                ((EQL (CAR l) e) (REMOVE e (CDR l)))
                (T (CONS (CAR l) (REMOVE e (CDR l))))
        )
)
```

SUBST subr 3 (e1 e2 e3): e

Returns a copy of **e3** in which all occurrences, at any level, of the expression **e2** have been replaced by **e1**. The expression **e3** is not changed.

```
(SUBST 'A 'B '(A B (B (D B F) B) B))
                is evaluated at (A A (A (D A F) A) A)
```

A possible implementation of the SUBST function is:

```
(DEFUN SUBST (e1 e2 e3)
        (COND
                ((EQUAL e2 e3) e1)
                ((ATOM e3) e3)
                (T (CONS (SUBST e1 e2 (CAR e3))
                                (SUBST e1 e2 (CDR e3))))
        )
)
```

REMOVE-IF nsubr 2 (e l): l

Returns a copy of list **l** in which all items that satisfy the test designated by **e** are removed. The expression **e** must designate a valid test function to be applied to the individual elements of list l. Its destructive variant is the DELETE-IF function.

- (REMOVE-IF 'NUMBERP ′(1 + 1 = 2)) is evaluated at (+ =)

- (DEFUN F (L) ; function with result T if the argument is list
```
    (COND
            ((ATOM L) NIL)
            (T T)
    )
)
```
- (SETQ L '(1 (2) (3 (4)))
- L is evaluated at (1 (2) (3 (4)))
- (REMOVE-IF 'F L) = (1) ; delete all elements of L that are lists

3. Examples of function definitions

We will further present definitions of Lisp functions for solving certain problems. We leave the reader to describe the recursive problem-solving formulas.

1. Define a function that determines the first atom in a list

a. at superficial level

```
(FIRST '((A B) ((1)) C (D E))) = C
```

```
(DEFUN FIRST (L)
        (COND
                ((NULL L) NIL)
                ((ATOM (CAR L)) (CAR L))
                (T (FIRST (CDR L)))
        )
)
```

b. at any level

```
(FIRST '(((A) B) ((1)) C (D E))) = A
```

```
(DEFUN FIRST (L)
        (COND
                ((NULL L) NIL)
                ((ATOM (CAR L)) (CAR L))
                (T (FIRST (CAR L)))
        )
)
```

2. Define a function that determines the sum of the first N elements in a linear list consisting only of numerical atoms.

```
        (SUM 3 '(1 2 3 4)) = 6
        (SUM 3 '(1 2)) = 3

    (DEFUN SUM (N L)
        (COND
            ((NULL L) 0)
            ((= N 0) 0)
            (T (+ (CAR L) (SUM (- N 1) (CDR L)))
        )
    )
```

3. Define a function that determines the sum of numeric atoms in a nonlinear list:

a. at superficial level

```
        (SUM '(1 (2 (C 3)) E 4)) = 5

(DEFUN SUM (L)
    (COND
        ((NULL L) 0)
        ((NUMBERP (CAR L)) (+ (CAR L) (SUM (CDR L))))
        (T (SUM (CDR L)))
    )
)
```

b. at any level

```
        (SUM '(1 (2 (C 3)) E 4)) = 10

(DEFUN SUM (L)
    (COND
        ((NULL L) 0)
        ((NUMBERP (CAR L)) (+ (CAR L) (SUM (CDR L))))
        ((ATOM (CAR L)) (SUM (CDR L)))
        (T (+ (SUM (CAR L)) (SUM (CDR L))))
    )
)
```

4. Define a function that returns the list of non-numerical atoms at any level in a nonlinear list:

a. without preserving the structure of the sublists.
(LIS '(1 A ((B) 6) (2 (C 3)) D 4)) = (A B C D)

```
(DEFUN LIS (L)
    (COND
        ((NULL L) NIL)
        ((NUMBERP (CAR L)) (LIS (CDR L)))
        ((ATOM (CAR L)) (CONS (CAR L) (LIS (CDR L))))
        (T (APPEND (LIS (CAR L)) (LIS (CDR L))))
    )
)
```

Do you see any problem with this version?

b. while maintaining the structure of the sublists.
(LIS '(1 A ((B) 6) (2 (C 3)) D 4)) = (A ((B)) ((C)) D)

```
(DEFUN LIS (L)
    (COND
        ((NULL L) NIL)
        ((NUMBERP (CAR L)) (LIS (CDR L)))
        ((ATOM (CAR L)) (CONS (CAR L) (LIS (CDR L))))
        (T (CONS (LIS (CAR L)) (LIS (CDR L))))
    )
)
```

5. Define a function that returns the list of pairs between an atom and the elements of a list.

```
(LPAIR 'A '(B C D)) = ((A B) (A C) (A D))
```

```
(DEFUN LPAIR (E L)
    (COND
        ((NULL L) NIL)
        (T (CONS (LIST E (CAR L)) (LPAIR E (CDR L))))
    )
)
```

6. Define a function that returns the list obtained by inserting an element at a certain position p in a list L ($1 \le p \le$ length of list L + 1).

```
(INS 'B 2 '(A C D)) = (A B C D)
(INS 'B 4 '(A C D)) = (A C D B)
```

```
(DEFUN INS (E N L)
    (COND
        ((= N 1) (CONS E L))
        (T (CONS (CAR L) (INS E (- N 1) (CDR L))))
    )
)
```

7. Define a function that returns the set of lists obtained by inserting an element on positions 1, 2,…, the length of the list L + 1 in a list L.

```
(INSERTION '1 '(2 3)) = ((2 3 1) (2 1 3) (1 2 3))
```

Remark: An auxiliary function (INSERT E N L) will be used which returns the set formed with the lists obtained by inserting an element on the positions N, N + 1,… the length of the list L + 1 in the list L.

```
(DEFUN INSERT (E N L)
    (COND
        ((= N 0) NIL)
        (T (CONS (INSERT E N L) (INSERT E (- N 1) L)))
    )
)

(DEFUN INSERTION (E L)
    (INSERT E (+ (LENGTH L) 1) L)
)
```

4. Binary trees

A binary tree can be memorized in the form of a list in the following two ways:

V1

A tree with

1. root
2. subtree-left and subtree-right

will be represented in the form of a nonlinear list of the form

(root list-subtree-left list-subtree-right)

where left-subtree list represents the list (represented as V1) associated to the left subtree of the root node list-subtree-right represents the list (represented as V1) associated to right subtree of the root node

For example the tree

```
    a
   /  \
  b    d
 \    /
  f  e
```

shall be represented, in **V1** variant, in the form of a list **(a (b () (f)) (d (e)))**.

Remark: If the left-right position of the unique subtree is relevant, then the empty subtree would have to be explicitely indicated. The minimal approach here is that the given subtrees are allocated from left to right, and whatever is missing afterwards is considered empty subtree. Another option is to explicitely show in the representation all the missing subtrees. The example above shall be represented thus as **(a (b () (f () ())) (d (e () ()) ()))**.

The **V1** representation is best suited for the list-form representation of a tree with a root, being appropriate to the recursive definition of a (binary) tree.

V2

A tree having

1. root
2. a number of nr-trees subtrees

shall be represented in the form of a linear list of the form

(root nr-subtrees
representation of subtree-1
representation of subtree-2
…)

where representation of subtree-i is the representation of the i-th subtree of root node as V2, while the whole construction is a linear list.

The disadvantage of the **V2** representation is that it is not suitable for a binary search tree (e.g. in the case of binary tree, no distinction is made between left and right subtrees).

For example the tree

```
    a
  /   \
 b     d
/\    /\
c  f   e  h
/  /
i  g
```

shall be represented, in the **V2** version in the form of the list

(a 2 b 2 c 1 i 0 f 1 g 0 d 2 e 0 h 0)

5. Examples of problems using trees

EXAMPLE 5.1 A binary tree represented in V1 is given. Determine the linear list of vertices obtained by traversing the tree inorder.

(inorder '(a (b () (f)) (d (e)))) = (b f a e d)

In the representation of a tree in V1 variant, we note thefollowing

1. (car l) – the first element of the list is the root of the tree
2. (cadr l) – the second element of the list, at superficial level, is the left subtree
3. (caddr l) – the third element of the list, at superficial level, is the right subtrees

Recursive model

$$inordine(l_1 l_2 l_3) = \begin{cases} \emptyset & daca\ l\ e\ vida \\ inordine(l_2) \oplus l_1 \oplus inordine(l_3) & altfel \end{cases}$$

```
(defun inorder (l)
        (cond
            ((null l) nil)
              (t (append (inorder (cadr l))
                    (list (car l)) (inorder (caddr l))))
        )
)
```

EXAMPLE 5.2 A binary tree represented in V2 is given. Determine the list of nodes obtained by traversing the tree inorder.

In the representation of a tree in V2, in relation to the V1 representation, the left and right subtrees are not clearly identified, so that the tree can be processed recursively.

__Idea:__ If we were able to extract the corresponding list (representation in V2) to the left and right sub-trees, the problem would be reduced to that shown in EXAMPLE 3.3.

We will use an auxiliary function to determine the left subtree of an tree represented in V2 (we assume the correct representation).

(stang '(a 2 b 2 c 1 i 0 f 1 g 0 d 2 e 0 h 0)) = (b 2 c 1 i 0 f 1 g 0)

We will use an auxiliary function, **parcurg_st**, which will go through the list starting with the 3rd element and that will return the left subtrees.
__Idea:__ collect the elements, starting with the 3rd element of the list, ... until when?

(parcurg_st '(b 2 c 1 i 0 f 1 g 0 d 2 e 0 h 0)) = (b 2 c 1 i 0 f 1 g 0)

$$stang(l_1\ l_2 \ldots l_n) = parcurg\_st(l_3 \ldots l_n, 0, 0)$$

nv – number of vertices
nm – number of edges

$$parcurg\_st(l_1\, l_2 \ldots l_k, nv, nm) = \begin{cases} \emptyset & daca\ l\ e\ vida \\ \emptyset & daca\ \ nv = 1 + nm \\ l_1 \oplus\ l_2 \oplus parcurg\_st(l_3 \ldots l_k, nv + 1, nm + l_2) & altfel \end{cases}$$

```
(defun parcurg_st(arb nv nm)
      (cond
     ((null arb) nil)
   ((= nv (+ 1 nm)) nil)
        (t (cons (car arb)
                      (cons (cadr arb)
                          (parcurg_st (cddr arb) (+ 1 nv) (+ (cadr arb) nm))
                    )
           )
        )
      )
)

(defun stang (arb)
        (parcurg_st (cddr arb) 0 0)
)
```

Homework. Write the function to determine the right subtree.

1. The same idea can be used as when going through the left subtree, but when nv=1+nm, the remaining list will be returned
2. To reduce the running time complexity, a certain function can be defined to return both the left and right subtrees.

```
    (drept '(a 2 b 2 c 1 i 0 f 1 g 0 d 2 e 0 h 0)  ) = (d 2 e 0 h 0)
```

Having the previous functions that determine the left and right subtrees, the list of vertices obtained by traversing the tree inorder will be constructed similarly to the recursive model described in **EXAMPLE 5.1.**

$$inordine(l_1\, l_2 \ldots l_n) = \begin{cases} \emptyset & daca\ l\ e\ vida \\ inordine(stang(l_1\, l_2 \ldots l_n)) \oplus\ l_1 \oplus inordine(drept(l_1\, l_2 \ldots l_n)) & altfel \end{cases}$$

EXAMPLE 5.3 Given a set represented in the form of a linear list, determine the list (set) of subsets of the list.

(subm '(1 2)) → (() (2) (1) (1 2))

How do we get the list of subsets of (1 2) if we know how to generate the list of subsets of the list (2)?

Idea: If the list is empty, its subset is the empty list. For the determination of the subsets of a list with the head E and the tail L, we will do the following:

1. determine a subset of the list L
2. place the element E in first position in a subset of list L

We will use an auxiliary function **insFirst** (*e*, *l*) which has as parameters an element *e* and a list *l* of linear lists and returns a copy of the list *l* in whose lists *e* is inserted on the first position.

(insFirst'3 '(() (2) (1) (1 2)) → ((3) (3 2) (3 1) (3 1 2))

; *l* is a list of linear lists
; insert *e* in the first position, in each list of *l*, and return the result

$$insPrimaPoz(e, l_1 l_2 \ldots l_n) = \begin{cases} \emptyset & daca\ lista\ e\ vida \\ (e, l_1) \oplus insPrimaPoz(e, l_2 \ldots l_n) & altfel \end{cases}$$

```
(defun insFirst(e l)
      (cond
            ((null l) nil)
```

```
        (t (cons (cons e (car l)) (insFirst e (cdr l))))
       )
)
```

; l is a mute represented in the form of a linear list

$$subm(l_1 \, l_2 \ldots l_n) = \begin{cases} (\emptyset) & daca\ lista\ e\ vida \\ subm(l_2 \ldots l_n) \oplus insPrimaPoz(l_1, subm(l_2 \ldots l_n)) & altfel \end{cases}$$

```
(defun subSet(l)
      (cond
              ((null l) (list nil))
              (t (append (subSet (cdr l))
                           (insFirst (car l) (subSet (cdr l)))
              ) )
       )
)
```

In the previously written Lisp code we note that the recursive call (subSet (cdr l) of the second COND clause is repeated, which obviously is not efficient from the perspective of time complexity. A solution to avoid this repeated call is to use an anonymous LAMBDA function (will be discussed in Lecture 10).

Also in Lecture 11 we will discuss simplifying the implementation of the subSet function, by giving up the use of the auxiliary function, using a MAP function.

EXAMPLE 5.4 A set represented in the form of a linear list is given. It is required to determine the list (set) of permutations of the given list.

(permutations '(1 2 3)) → ((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1))

How do we get the list of permutations of (1 2 3) if we know how to generate the list of permutations of the list (2 3)? ((2 3) (3 2))

Idea: If the list is empty, the list of its permutations is the empty list. For determining the permutations of a list [E | L], which has head E and tail L, we will do the following:

1. get a permutation L1 of the L list;
2. place element E on all positions of the L1 list and thus produce the X list which will be a permutation of the initial list [E | L].

We will use some auxiliary functions:

(1) a function that returns the list obtained by inserting an **element E** on a specific N position in an **L** list (1 ≤ N ≤ length of list **L**+1).

(INS '1 2 '(2 3)) = (2 1 3)
(INS '1 3 '(2 3)) = (2 3 1)

<u>Recursive model</u>

$$ins(e,n,l_1\, l_2 \ldots l_k) = \begin{cases} (e\;\, l_1\, l_2 \ldots l_k) & daca\; n=1 \\ l_1 \oplus ins(e, n-1, l_2 \ldots l_k) & altfel \end{cases}$$

```
(DEFUN INS (E N L)
      (COND
          ((= N 1) (CONS E L))
        (T (CONS (CAR L) (INS E (- N 1) (CDR L))))
        )
)
```

(2) a function to return the set of lists obtained by inserting an **element E** on positions **1, 2,..., the length of the L+1 list** in a list **L**.

(INSERTION '1 '(2 3)) = ((2 3 1) (2 1 3) (1 2 3))

Note: An auxiliary function (INSERT E N L) will be used that returns the set formed with the lists obtained by inserting an item on positions **N, N-1, N-2,....,1** in the list **L**.

(INSERT '1 2 '(2 3)) = ((2 1 3) (1 2 3))

$$insert(e,n,l_1\, l_2 \dots l_k) = \begin{cases} \emptyset & daca\ n = 0 \\ ins(e,n,l_1\, l_2 \dots l_k) \oplus insert(e,n-1,l_1 l_2 \dots l_k) & altfel \end{cases}$$

```
(DEFUN INSERT (E N L)
        (COND
                ((= N 0) NIL)
                (T (CONS (INS E N L) (INSERT E (- N 1) L)))
        )
)
```

$$inserare(e, l_1 l_2 \ldots l_n) = insert(e, n+1, l_1 l_2 \ldots l_n)$$

```
(DEFUN INSERTION (E L)
 (INSERT E  (+ (LENGTH L) 1)  L)
)
```

Homework Continue the implementation of the permutations function.

EXAMPLE 5.5. Given a numerical linear list, return the list sorted using tree sorting.

(sortare '(5 1 4 6 3 2)) = (1 2 3 4 5 6)

For tree sorting of a list, we will do the following:

1. We build an intermediate binary search tree with the elements of the initial list
    1. for memorizing the tree we will use a linear list – variant V1
    2. for the construction of the intermediate tree we will need to insert an element into the tree
2. Going through the previously built tree with inorder traversal will provide us with the original list correctly sorted.

<u>Recursive model</u> – inserting an element into the tree

$$inserare(e, l_1 l_2 l_3) = \begin{cases} (e) & daca\ l\ e\ vida \\ l_1 \oplus inserare(e, l_2) \oplus l_3 & daca\ e \le l_1 \\ l_1 \oplus l_2 \oplus inserare(e, l_3) & altfel \end{cases}$$

```
(defun inserare (e tree)
        (cond
                ((null tree) (list e))
                ((<= e (car tree)) (list (car tree)
                        (inserare e (cadr tree)) (caddr tree)))
                (t (list (car tree) (cadr tree) (inserare e (caddr tree))))
        )
)
```

Recursive model – building the tree from the list

$$construire(l_1 \, l_2 \ldots l_n) = \begin{cases} \emptyset & daca \; l \; e \; vida \\ inserare(l_1 construire(\, l_2 \ldots l_n)) & altfel \end{cases}$$

```
(defun construire (l)
        (cond
                ((null l) nil)
                (t (inserare (car l) (construire (cdr l))))
        )
)

; list of nodes of a tree traversed inorder
(defun inordine (arb)
        (cond
            ((null arb) nil)
                (t (append (inordine (cadr arb))
                        (list (car arb))
                        (inordine (caddr arb))))
        )
)

; tree sorting of the list
(defun sort (l)
        (inordine
                (construire l)
        )
)
```

What is the time complexity of tree sorting?

Homework Consider given a binary tree, whose nodes are distinct, represented in the form of a nonlinear list of the form (root list-subtree-left list-subtree-right). Return a linear list representing the path from the root to a given node.

For example:

- (setq arb '(a (b () (f)) (d (e () (k)) (l))))

```
(cale 'm arb) → NIL
(cale 'f arb) → (a b f)
(cale 'k arb) → (a d e k)
```

```
      a
    /   \
   b     d
   \    /  \
    f   e   l
         \
          k
```

$$cale(e, l_1\, l_2 l_3) = \begin{cases} \emptyset & daca\ l\ e\ vida \\ (e) & daca\ e = l_1 \\ l_1 \oplus cale(e, l_2) & daca\ e \in l_2 \\ l_1 \oplus cale(e, l_3) & daca\ e \in l_3 \\ \emptyset & altfel \end{cases}$$