# Functional Programming – Laboratory 8
Surgery, Lambda expressions, Circular lists, Mapping

Isabela Drămnesc

April 18, 2012

## 1 Concepts

- append, nconc
- reverse, nreverse
- remove, delete
- remove-if, delete-if
- lambda expressions
- apply, funcall
- mapcar, maplist

## 2 Questions from Laboratory 6

- What forms of DO do you know? Give some examples to show the functionality of each form!
- What is the difference between append and nconc? Represent using cons cells the behavior of nconc. Define your own function nconc.
- What is the difference between reverse and nreverse? Represent using cons cells the behavior of nreverse. Define your own function nreverse.
- Write the iterative definition in Lisp for:
  1. gcd(a,b)
  2. factorial(n)
  3. my-reverse(my-list)
  4. length(my-list)

## 3 Exercises

### 3.1 remove, delete

```
> (setq l '(today is sunny))

> (remove 'sunny l)

> l

> (delete 'sunny l)

> l
```

Remark:

When using the “surgery” function, even if the result is correct, the side effect does not appear in the following situations:

- when we use nconc and the first list is nil;
- when we use delete and we delete the first element from a list;

Exemple:

```
> (setq *l* nil)

> *l*

> (nconc *l* '(a b c))

> *l*

> (setq *ll* '(a b c d))

> (delete 'a *ll*)

> *ll*

> (delete 'b *ll*)

> *ll*
```

### 3.2 remove-if, delete-if, subst

```
> (subst 'a 'b '(a b (b (d e f) b) b))

> (setq w '(1 + 1 + 1 = 3))

> w

> (remove-if 'numberp w)
> w

> (delete-if 'numberp w)

> w
```

### 3.3 Lambda expresions

We use them:

- when a function is used only once and it is to simple to write a separate definition for it;
- when the function has to be applied dynamically (is impossible to define it using DEFUN).

Syntax:
(lambda l f1 f2 f3 ... fn)
or
((lambda l f1 f2 f3 ... fn) par1 par2 ... parn)

- defines a function used locally;
- l represents the list of parameters; (can be given explicitly (a fixed number of parameters) or the list can have a variable number of parameters);
- f1 f2 ... fn the body of the function;

The arguments are evaluated when the function is called. If we want the arguments not to be evaluated when the function is called, then we have to use QLAMBDA.

Exemple:

```
> ((lambda () 20))

> ((lambda (x) (1+ x)) 5)

> ((lambda (x y) (+ x y)) 5 7)

> ((lambda (y)
   ((lambda (x) (+ x (* 2 y)
                     (* 12 x y)
                     (* (* x x) (* y y))
                  )
     )
     21
    )
   )
   2
   )

> ((lambda (x y)
        (+ x
              (* 2 y)
```

```
                  (* 12 x y)
                  (* (* x x) (* y y))))
 21 2)
```

Lambda expressions can be considered anonymous functions (without name).
Example:

```
(setf (symbol-function 'add_with_3)
      '(lambda (x) (+ x 3)))

>(add_with_3 26.6)
```

## 3.4 apply, funcall

There are cases when the number of parameters of a function has to be set dynamically. Applying a function on a set of parameters possibly dynamically synthesized is possible using the functions APPLY and FUNCALL.

Sintax:
(apply function (list-of-parameters))

Examples:

```
> (apply 'cons '(a b))

> (apply 'max '(1 2 3 4 5 6))

> (apply '+ '(1 2 3 4))
```

Sintax:
(funcall function arg1 arg2 ...)

Is a version of the function apply which allows the application of a function to a fixed number of parameters.
Examples:

```
> (funcall 'cons 'a 'b)

> (funcall 'max 1 2 3 4 5 6)

> (setq p 'car)

> ((eval p) '(a b c))
error: bad function - (EVAL P)   ; incorrect
     ;becomes correct if we use apply or funcall

> (setq p 'car)

> p

> (funcall p '(a b c))

> (apply p '((a b c)))
> (setq f '(lambda(x) (+ x 3)))

> f

> (f 6)

> (funcall f 6)
```

## 3.5 Dynamically modification of functions

Because the functions are represented as data, lists, they can be dynamically modified during the program as any data. Example:

```
> (setq f '(lambda(x) (+ x 3)))

> (funcall f 5)

> (setf (third (third f)) 4)

> f

> (funcall f 5)
```

## 3.6 mapcar, maplist

MAPCAR is a function with global application. It is applied on each of the arguments from the list of arguments, the results are put into a list and returned when calling the function MAPCAR. The evaluation is done when we reach the end of the shortest list.

```
> (mapcar 'oddp '(1 2 3))
(T NIL T)

> (mapcar 'list '(1 2 3))
((1) (2) (3))

> (mapcar '+ '(1 2 3) '(4 5 6))
(5 7 9)

> (mapcar 'cons '(1 2 3) '(4 5 6))
((1 . 4) (2 . 5) (3 . 6))

> (maplist 'length '(1 2 3))
(3 2 1)

> (mapcar 'list '(a b c) '(1 2))
((A 1) (B 2))

> (mapcar 'car '((a b c) (x y z)))
```

```
(A X)
```

MAPLIST is applied to the whole lists, then to the cdr of the lists, then to the cddr of the lists (to all the successive sublists) until one of them is nil. The result is a list containing the successive results obtained.

```
> (maplist 'append '(a b c) '(1 2 3))
((A B C 1 2 3) (B C 2 3) (C 3))

> (maplist '(lambda (x) x) '(a b c))
((A B C) (B C) (C))

> (maplist 'list '(a b c))
(((A B C)) ((B C)) ((C)))

> (maplist 'length '(1 2 3))
(3 2 1)
```

## 3.7 Circular lists

Study the following example:

```
(defun list-len (x)
  (do ((n 0 (+ n 2))                ;Counter
       (fast x (cddr fast))         ;Fast pointer: goes from 2 to 2
       (slow x (cdr slow)))         ;Slow pointer: passes through each cdr
      (nil)
      ;; If the fast pointer reaches the end, then will return n.
      (when (endp fast) (return n))
      ;; If cdr of the fast pointer reaches the end, then returns n+1.
      (when (endp (cdr fast)) (return (+ n 1)))
      ;; If the fast pointer catches up the slowly pointer,
      ;; means that we deal with a circular list.
      ;; We return nil.
      (when (and (eq fast slow) (> n 0)) (return nil))))
```

Test also the following example:

```
(setq x '(1 2))

> (rplacd x x)

> (list-len x)
```

## 3.8 Our own equality predicate

```
A function which decides the "equality" of two structures
constructed with cons:

(defun our-equal (l1 l2)
  (cond ((and (null l1) (null l2)) t)
        ((and (atom l1) (atom l2)) (equal l1 l2))
        ((and (atom l1) (not (atom l2))) nil)
        ((and (not (atom l1)) (atom l2)) nil)
        ((our-equal (car l1) (car l2)) (our-equal (cdr l1) (cdr l2)))
        (t nil)
  )
)
```

## 4 Homework (deadline: next lab)

1. Evaluate the following s-expressions:

```
> (cadr '(a b c d e))

> (second '(a b c d e))

> (nth 2 '(a b c d e))

> (cadr (cadr '((a b c) (d e f) (g h i))))

> (cadadr '((a b c) (d e f) (g h i)))

> (cddadr '((a b c) (d e f) (g h i)))

> (last '((a b c) (d e f) (g h i)))

> (third '((a b c) (d e f) (g h i)))

> (cdr (third '((a b c) (d e f) (g h i))))
```

2. Define an iterative function RANGE with has as parameters a list of numbers and returns a list with the length 2 and the returned list contains the minimum element and the maximum element. Use the predicates > and < Ensure that your algorithm is linear. Write another function VALID-RANGE, which returns the same result as RANGE if the elements of the list are all numbers and INVALID otherwise.
Exemplu:

```
> (range '(0 7 8 2 3 -1))
(-1 8)
> (range '(7 6 5 4 3))
(3 7)
> (valid-range '('a 7 8 2 3 -1))
INVALID
> (valid-range '(0 7 8 2 3 -1))
(-1 8)
```

3. Write 2 s-expressions in order to access the symbol C for each of the following lists:

```
(a)   (A B C D E)

(b)   ((A B C) (D E F))
```

(c) ((A B) (C D) (E F))

4. Change C in SEE for each of the following lists (without using subst):

(a) (A B C D E)

(b) ((A B C) (D E F))

(c) ((A B) (C D) (E F))

(d) (A (B C D) E F)