Functional Programming – Laboratory 2

Define new functions

Isabela Drămnesc

February 29, 2012

1 Concepts

- Local and global variables, constants
- Assignment, Equalities
- If, Cond
- Define new functions

2 Questions from laboratory 1

- What is the difference between cons, list, append?
- How we represent lists in LISP? Draw the box notation for the list (today (is 1) March)
- What will return: > (cdar '((a (b c)) d ((e f) g) h))

3 Assignment, Global variables, Constants

defvar (define variable) is used:
(defvar <variabile-name> [<initial-value> [<documentation>]])
defvar does not evaluate the expression <initial-value> only when the variable it is used and only if the variable has a value.

```
>(defvar y 10)

>y

> (defvar z 3 "Define a variable")

> z

> (defvar z 2 "Define a variable")

> z                          ; the value is not changed
```

```
> (defvar z 0 "Define a variable")

> z

> (defvar z "Define any variable")

> z

> (defvar z 8 "Define any variable")

> z
        ;z still has the initial value when it was defined

> (defvar w 8 "Define any variable")

> w

> (defvar *-name-* nil "This variable is called -name-")

> (defvar *maxim* (max z w))

> *maxim*
```

defparameter (define parameter) is used for creating a global variable:
(defparameter <param-name> <initial-value> [<documentation>])

```
> (defparameter par "necessary value")

> par

> (defvar par 4)

> par

> (defvar m 6)

> m

> (defparameter m 9)

> m

> (defvar m 5)

> m

> par
```

defconstant (define constant) is used:
(defconstant <constant-name> <value>)

```
> (defconstant *pi* 3.14159265358979323)

> (defconstant *pi* 2)

> *pi*
```

setq is a special form and it is used:
(setq <variable-1> <expression-1> <variable-2> <expression-2> ...)
setq evaluates <expression-1>, assigns to <variable-1> the result of the evaluation, then it goes to the next pair expression-value...
At the end it will return the result of the last expression that was evaluated.

```
> (setq d '(a b c))   ;setq assigns a value to a symbol
                      ;setq is an exception to the evaluation rule
                      ;the first argument is not evaluated q=quote
                      ;d is a global variable

> d

> (setq 'a 1 'b 2)

> (setq a 1 'b 2)

> (setq x (+ 7 3 0) y (cons x nil))

> (setq h (+ 2 3 4) i '(p o m))

> h

> i

> (setq)

> (setq j)

> (setq j 1)

> (setq k 2)

> (psetq j k k j)    ;psetq works like setq,
                     ;but assignments are made in parallel
> j

> k
```

set is a function and it is used:
(set <variable-1> <value-1> <variable-2> <value-2> ...)
evaluates all the arguments and return the result of the evaluation of the last argument, and as a side effect <variable-i> evaluates to <value-i>.
(set 'x 2) ≡ (setq x 2)

```
> (set 'q 0)
```

```
> (set 'y 'x)

> y

> x

> (setf x 10)     ;setf assigns a value to the global variable x
                  ;and returns the value assigned

> (setf y (reverse '(come has March)))

> x

> y

> (setf g '(1 2 3) f (+ 2 3 4))

> g

> (setq x '(a b c))

> (setf (car x) 'k) ;the first argument of setf can be an expression

> x

> (setq y '(1 2 3 4))

> y

> (setq (car y) 10)

> (setf (car y) 10)

> y
```

4 Equalities in LISP

$$\begin{pmatrix} x & y & eq & eql & equal & equalp \\ 'x & 'x & T & T & T & T \\ '0 & '0 & ? & T & T & T \\ '(x) & '(x) & nil & nil & T & T \\ '"xy" & '"xy" & nil & nil & T & T \\ '"Xy" & '"xY" & nil & nil & nil & T \\ '0 & '0.0 & nil & nil & nil & T \\ '0 & '1 & nil & nil & nil & nil \end{pmatrix}$$

```
; Two objects are EQ if they have the same memory location.
```

```
; EQ is the strongest equality.
; Symbols are EQ
>(eq 'a 'a)
>(eq nil nil)
>(eq nil (cdr '(a)))
>(eq t t)
>(setq x 'b)
>(setq y 'b)
>(eq x y)
; lists are EQ if they have the same memory location
> (setq c '(f 2 . 3))
> (setq d '(f 2 . 3))
> (eq c d)
> (eq c c)
> (eq d (car (list d c)))
> (eq (car c) (car d))
>(eq 2.3 2.3)
>(eq 2 2)
>(eq 2012 2012)
> (eq #c(1 2) #c(1 2))
> (eq 1/2 1/2)
>(eq (car (cdr c)) (car (cdr d)))
>(eq (car (cdr c)) (car (cdr c)))
;EQL
;for lists is the same as EQ
```

```
>(eql '(a b) '(a b))
>(eql (cons 'a nil) (cons 'a nil))
>(setf x (cons 'a nil))
>(eql x x)
>(eql 'a 'a)
>(eql (cddr a) (cddr b))
>(eql (cdddr a) (cdddr b))
>(cdddr a)
; for numbers: the same type and the same value
>(typep 'ztesch 'symbol)
>(typep 3 'symbol)
>(typep 'ztesch 'number)
>(typep 3 'number)
>(typep 'ztesch 'atom)
>(typep 3 'atom)
>(atom 3)
>(atom 3.3)
>(atom a)
>(atom 'a)
>(typep -3.57 'fixnum)
>(typep 333333333333333333333 'bignum)
>(symbolp 'a)
>(symbolp 3)
>(symbolp '(x y))
>(stringp "a string")
```

```
>(stringp '(a string))

>(eql 'a 'a)

>(eql 2 2)

>(eql 2.2 (+ 0.7 1.5))

>(eql 2 2.0)

>(eql 2 2.)

>(eql "string" "string")

; EQUAL
; returns true if their arguments print the same

>(equal x (cons 'a nil))

>(equal '(a b c) '(a . (b . (c . nil))))

> (eql '(a b . c) (cons 'a (cons 'b 'c)))

> (equal '(a b . c) (cons 'a (cons 'b 'c)))

> (setf k1 '(a b . c))

> (setf k2 (cons 'a (cons 'b 'c)))

> (eql k1 k2)

> (equal k1 k2)

> (setf x '(a b c))

> (setf y x)

> (eql x y)

> (eq x y)
```

Conclusions:

- EQ: returns T if their arguments are the same objects and NIL otherwise;
- EQL: two elements are EQL if they are EQ or if they are numbers of the same type;
- EQUAL: returns T if the values of their arguments are equivalent S-expressions (if the two S-expressions have the same structure, example: two lists have the same elements);

5 Predicates with more than one argument

```
> (> 77 10)

> (> 10 77)

> (>= 25 25)

> (<= 25 25)

> (>= 100 6)

> (>= 6 100)

> (< 1 2 3 4 5 6 7 8 9 10 11 13 17)

> (< 1 2 3 4 5 6 7 8 9 10 19 13 17)
```

6 If, Cond

if is used:
(if <test> <then-expression> [<else-expression>])

```
> (setq x 21)

> (if (= x 21)
      (print 'today))

>(if (> 3 2) (+ 4 5) (* 3 7))

> (if (< 3 2) (+ 4 5) (* 3 7))

> (if (+ 2 3) 1 2)

> (equalp (+ 2 3) t)

> (if nil 1 2)

>(if (atom 'x) 'yes 'no)

>(if (atom (5 6)) 'yes 'no)

>(if (atom (+ 5 6)) 'yes 'no)

>(if (atom '(5 6)) 'yes 'no)

>(* 5 (if (null (cdr '(x)))
       0
       (+ 11 12)))
```

```
>(* 5 (if (null (cdr '(x y)))
        0
        (+ 11 12)))
```

cond is used:

```
(cond
    (<test_1> <consequence_1_1> <consequence_1_2> ...)
    (<test_2>)
    (<test_3> <consequence_3_1> ...)
    ...
)
```

```
> (cond ((= 2 3) 1)
        ((< 2 3) 2)
   )
```

```
> (cond ((= 2 3) 1)
        ((> 2 3) 2)
        (t 3)
   )
```

```
> (cond ((= 2 3) 1)
        ((> 2 3) 2)
        (3)
  )
```

```
> (cond ((= 2 2) (print 1) 8)
        ((> 2 3) 2)
        (t 3)
  )
```

We ask like this:

```
(cond (x 'b)
      (y 'c)
      (t 'd)
)
```

If $x = t$? (then return b)

If $x = nil, y = t$? (then return c)

If $x = nil, y = nil$? (then return d)

Another example:

```
(cond (x (setf x 1) (+ x 2))
      (y (setf y 2) (+ y 2))
      (t (setf x 0) (setf y 0))
)
```

If $x = t$? (then return 3) Cine e x?($x = 1$)

If $x = nil, y = t$? (then return 4) Who are x and y? (nil and 2)

If $x = nil, y = nil$? (then return 0) Who are x and y? (0 and 0)

7 User's functions

7.1 Define new functions

(defun <function-name> <parameter-list>
<expr-1> <expr-2> ... <expr-n>)
where:
<function-name> is the first argument and represents the name of the function that is defined by defun;
<parameter-list> is the second argument of defun, is of the form (<par-1> <par-2> ... <par-m>) and represents the list of the parameters if the defined function;
<expr-i>, i = 1, . . . , n are the body of the defined function.

```
> (defun triple (x)
                (* 3 x))

> (triple 4)

> (triple 50)

> (triplu (50))

> (defun my-sum (x y)
            (+ x y)
  )

> (my-sum 3 5)

> (defun is-even-number (x)
              (equal (mod x 2) 0)
  )

> (is-even-number 6)

> (is-even-number 7)

> (defun even-number-or-divisible-with-7 (x)
            (or (is-even-number x) (equal 0 (rem x 7)))
  )

> (even-number-or-divisible-with-7 7)

> (even-number-or-divisible-with-7 0)
```

```
> (even-number-or-divisible-with-7 10)

> (even-number-or-divisible-with-7 11)

> (even-number-or-divisible-with-7 14)

> (setq i 10)

> (triple i)

> (defun my-third (lista)
          (car(cdr(cdr lista)))
   )

> (my-third '(r t y))

> (my-third '(r t y h))

; A confusing example

>(defun conf (list)
        (list list)
 )

>(conf 5)

>(conf 5 6)

>(conf (5 6))

>(conf '(5 6))     ; Explain!

>(length '(c b a))

; Define our own function "len" which returns the length of a list

>(len '(c b a))

>(defun len (list)
    (+ 1 (len (cdr list))
    )
 )

>(len '(1 2 3 4))

;;; the base case is missing: infinite loop

(trace len)
```

```
(len '(1 2 3 4))

;;; infinite loop: stop with CTRL C

;;; define len correctly -- a recursive function

>(defun len2 (lst)
  (if (null lst)
        0              ;; base case
   (+ 1 (len2 (cdr lst))) ;; inductive case
  )
 )

>(len2 '(1 2 3 4))

>(untrace)

>(len2 '(1 2 3 4))

>(len2 ())

>(len2 '(a))

>(len2 '(b a))

>(len2 '(c b a))

>(len2 '((a b c d e f) g h (i j) k))

>(len2 (car '((a b c d e f) g h (i j) k)))

; A function with two parameters which
;returns T if the length of the first list is greater than
;the length of the second list

> (defun longer-listp (list1 list2)
(if
          (> (length list1) (length list2))
           T
           nil
        )
)

> (longer-listp '(1 2 3) '(5 6))

> (longer-listp '(1 2 3) '(5 6 l l l l))
```

7.2 Exercises

1) Write a function which returns x power y.

2) Write a function which returns $x * x * y * y$.

3) Write a function which returns the number of numbers which occur in a list.

4) Write a function SAME-ELEMENTS which has an argument a list and returns T if all the elements from a list are equal and NIL otherwise.

8 Homework - deadline: next lab

Problem 1

Write the following s-expressions in Lisp. What is the result for each case?

```
> (setq a '(u v w))

>(set (car (cdr a)) 'b)

> a

> (setq a '(u v w))

> '(setq a '(u v w))

> a

> (setq a 'a)

> (setq b 'a)

> (list a b 'b)

> (list (list 'a 'b) '(list 'a 'b))

> (defun double (x)
            (* 2 x))

> (double 2.3)

> (defun times-square (x y)
            (* x y y))

> (times-square 4 3)

> (defun times-cube (x y)
        (* x y y y))

> (defun cube-times (x y)
```

```
        (times-cube y x))

> (cube-times 3 2)
```

Problem 2

Evaluate the following s-expressions:

```
> (zerop '3)

> (zerop 3)

> (atom 3)

> (null '(a b))

> (numberp '(a b))

> (consp '(a b))

> (listp '(a b))

> (consp nil)

> (not (null 'nil))

> (not (null 3))

> (not (atom  '(a b)))

> (not (atom 'a))

> (not (null '(a b)))

> (not (zerop 3.3))

> (not (zerop 0.0))

> (not (numberp 'b))
```

Problem 3

Write a function which has the parameters two lists and which returns the concatenation[1] of the two lists.

Problem 4

Write a function which calculates the factorial of a natural number.

[1] Please do not use the function append