Functional Programming – Laboratory 9

Optional arguments, Association lists

Isabela Drămnesc

April 25, 2012

1 Concepts

- &optional, &rest, &key
- Association lists.
- Properties.

2 Questions from Laboratory 8

3 Mandatory and optional arguments

Lambda expressions or a expressions using defun can have multiple types of arguments:

- mandatory arguments (at the beginning of the list of parameters);
- optional arguments (&*optional*);
- a rest argument (&*rest*);
- key arguments (&*key*);
- local variables (&*aux*).

Exemple:

3.1 Mandatory arguments:

```
> (defun f (x y)
      (list x y))

> (f 10 20)

> (f 10) ; Error

; x and y are mandatory arguments
; they must have a correspondence when calling the function
```

3.2 Mandatory and optional arguments:

```
> (defun f (x y &optional z v)
      (list x y z v))

> (f 10)

> (f 10 20)

> (f 10 20 30)

> (f 10 20 30 40)

> (f 10 20 30 40 50)

; z and v are optional arguments, if their
; correspondent is missing in the function call, then
; Lips uses NIL instead

; changing the implicit value NIL of the optional parameters
; Example:

> (defun f (x y &optional z (v (+ x y)))
      (list x y z v))

> (f 10 20)

> (f 10 20 30)

> (f 10 20 30 40)

;; if for an optional argument we also specify a variable <svar>
;; then this will be T if has a value specified for it
;; and NIL otherwise

(defun f (x y &optional z (v (+ x y) da?))
      (list x y z v da?))

> (f 10 20)

> (f 10 20 9)

> (f 10 20 9 4)

> (f 10 20 9 30)

; another example of setting implicit values

(defun f (x y &optional (z 11) (v 9))
      (list x y z v))
```

```
> (f 10 20)

> (f 10 20 9)

> (f 10 20 9 10)
```

3.3 &*rest* arguments

```
;;; a rest argument is used for functions which have a
; variable and a unknown number of arguments
; the variable that is after rest is bound to the variables remaining after
; binding the others
; A function can have only one rest argument

(defun f (x y &rest r)
       (list x y r))

> (f 4 5)

> (f 4 5 8)

> (f 4 5 8 10 29 88 -100 -99 203)


> (defun f (x y &optional z v &rest r)
       (list x y z v r))

> (f 3 4)

> (f 3 4 5)

> (f 3 4 5 6)

> (f 3 4 5 6 7)

> (f 3 4 5 6 7 8 9 10 11 12)

; whet is the effect of the following function?

> (defun my-add (nr &rest i)
         (cond ((null i) (+ nr 1))
               (t (+ nr (apply '+ i)))
         )
)

> (my-add 10)

> (my-add 10 2)
```

```
> (my-add 10 5)

> (my-add 10 5 8 9 11)
```

3.4 *&key* arguments

In the call they are specified as pairs keyword and value.

In the call the keywords have in front the character ":"

The advantage (even if they are optional) is that they can appear in any order or in any position in the call.

```
> (defun f (x y &key a b)
      (list x y a b))

> (f 10 20)

> (f 10 20 :a 11)

> (f 10 20 :a 11 :b 22)

> (f 10 20 :b 10)

> (f :r 10 :a :b)

> (f :a 10 :b 9)

> (f 22 10 :c 9)

;; another example

> (defun f (x &optional (y 3) &rest r &key c (d x))
      (list x y r c d))

> (f 4)

> (f 4 5)

> (f :c 9)

> (f 6 7)

> (f 6 7 :c 8)

> (f 6 7 :d 8)

> (f 6 7 :c 8 :d 9 :d 10)

> (f 6 7 :c 8 :c 9 :c 10 :c 11)
```

4 Association lists

The elements of an association list are cons cells where the corresponding of car is called key and the corresponding of cdr is called data. In order to introduce or to extract elements we operate on the extremities of the list, the behavior is similar for stack.

If we introduce a new pair key-data having an existing key, then "shading" of the old association is produced. After we eliminate the new one, the old one is "recovered". On this behavior is based the "binding" of the variables to values.

```
(setq things '((phone pocket)
               (money card)
               (book bag)
               (mark student_card)))

> (assoc 'book things)

> (assoc 'mark things)

> (assoc 'map things)

;; adding a new association

> (setq things (cons '(coat closet) things))

> (assoc 'coat things)

;; to gain space use dotted pairs

(setq things '((phone . pocket)
               (money . card)
               (book . bag)
        (money . pocket2)
               (mark . student_card)))

> (assoc 'money things)

;; in order to find money one solution is to use:

(setq things '((phone . pocket)
               (money_euro . card)
               (book . bag)
        (money_ron . pocket2)
               (mark . student_card)))

> (assoc 'money_euro things)

> (assoc 'money_ron things)

> (rassoc 'card things)   ;;; it works only for dotted lists
```

The access functions for association lists are:

- *assoc* to access the key of the searched element;
- *rassoc* for access the data;

Another example:

```
> (setq my-values (pairlis '(A B C) '(1 2 3)))

> my-values

> (assoc ' a my-values)

> (rassoc 1 my-values)
```

The *pairlis* function organizes an association list with the keys from a list and the data from other list. There is no order when introducing elements in an association list. Of course, the two list-argument have to have the same length.

```
> (pairlis (list 'a 'b 'c) (list 1 2 3))

> (pairlis '(A B C) '(1 2 3) '(23 2))
```

Locations and the access of locations:

```
> (setq the_list '(a b c))

> (setf (car the_list) 10 (cadr the_list) 'something)

> the_list
```

4.1 Problem

Write a function in Lisp which returns the list of all keys from an association list.

4.2 Problem

Write a function in Lisp assoc-all which returns the sublists of all the occurrences of a certain key in an association list.

4.3 Properties lists p-lists

Example1: (value-function)

```
> (setq s 10)

> s

> (symbol-value 's)

> (defun s (x) (+ x 2))
```

```
> (s 9)

> (symbol-function 's)

> (symbol-value 's)
```

Example2: The properties list of a symbol:

```
> (setf (get 's 'p1) 'propr1)

> (setf (get 's 'p2) 'propr2)

> (get 's 'p1)

> (get 's 'p2)

> (symbol-plist 's)

> (symbol-name 's)

> (symbol-package 's)

> (describe 's)

S is the symbol S, lies in #<PACKAGE COMMON-LISP-USER>, is accessible in 1
package COMMON-LISP-USER, a variable, value: 10, names a function, has 3
properties P2, P1, SYSTEM::DEFINITION.
For more information, evaluate (SYMBOL-PLIST 'S).

 #<PACKAGE COMMON-LISP-USER> is the package named COMMON-LISP-USER. It has 2
 nicknames CL-USER, USER.
 It imports the external symbols of 2 packages COMMON-LISP, EXT and exports
 no symbols, but no package uses these exports.

 10 is an integer, uses 4 bits, is represented as a fixnum.

 #<FUNCTION S (X) (DECLARE (SYSTEM::IN-DEFUN S)) (BLOCK S (+ X 2))> is an
 interpreted function.
 Argument list: (X)
```

Another example:

```
> (setq program nil)

> (setf (get 'program 'language) '(lisp prolog))

> (get 'program 'language)
(LISP PROLOG)

> (setf (get 'program 'variant) '("CLisp" "Quintus Prolog"))
```

```
> (get 'program 'variant)

> (setf (get 'program 'version) '(2.30 1.2))

> (get 'program 'version)

> program

> (get 'program 'operating_system)

> (setf (get 'sky 'color) 'blue)

> (get 'sky 'color)

> (remprop 'sky 'color)

> (get 'sky 'color)
```

5 Homework

Create an association list "*my_house*". This house contains multiple rooms and in each room we can find at least one object. Example: (lobby stand) (*dining_room* table chair1 chair2 TV couch) and so on.

Add and delete properties of objects. For example: the table from the *dining_room* is from wood and has a certain size.

Write a function *the_plan_of_the_house* which prints the plan of the house. Example: "We are in the house. First we enter the lobby, we let the jacket on the stand, then we enter the *dining_room* where we can see a wood table of size..., chair1, chair2, ..." and so on.