Programare funcţională – Laboratorul 9

Argumente opţionale, Liste de asociere

Isabela Drămnesc

April 24, 2012

1 Concepte

- &optional, &rest, &key
- liste de asociere.
- proprietăţi.

2 Întrebări din Laboratorul 8

3 Argumente obligatorii şi opţionale

Expresiile lambda sau un defun pot avea mai multe tipuri de argumente:

- argumente obligatorii (care se ragăsesc la începutul listei de parametri);
- argumente opţionale (&*optional*);
- un argument rest (&*rest*);
- argumente cheie (&*key*);
- variabile locale (&*aux*).

Exemple:

3.1 Argumente obligatorii:

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

> (f 10 20)

> (f 10) ; Error

; x si y sunt argumente obligatorii
; trebuie neaparat sa aiba corespondente in apelul de functie
```

3.2 Argumente obligatorii şi opţionale:

```
> (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 si v sunt argumente optionale, daca lipseste
; corespondentul lor in apelul de functie, atunci
; se utilizeaza NIL in locul lor

; pentru a schimba valoarea implicita NIL pentru parametri optionali
; Exemplu:

> (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)

;; daca specificam pentru un argument optional si o variabila <svar>
;; pe langa forma de initializare, aceasta va fi T daca s-a specificat o
;; valoare pentru argumentul respectiv, daca nu NIL

(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)

; alt exemplu setare valori implicite

(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 Argument *&rest*

```
;;; argumentul de tip rest este folosit pentru functiile cu un
; numar variabil si necunoscut de argumente la definitie
; variabila dupa rest este legata la variabilele ramase dupa legarea
; celorlalte argumente
; O functie poate avea doar un argument de tip rest

(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)

; ce efect are urmatoarea functie?

> (defun aduna (nr &rest i)
         (cond ((null i) (+ nr 1))
               (t (+ nr (apply '+ i)))
         )
)

> (aduna 10)

> (aduna 10 2)
```

```
> (aduna 10 5)

> (aduna 10 5 8 9 11)
```

3.4 Argumente de tip cuvinte cheie *&key*

La apel sunt precizate prin perechi cuvânt cheie şi valoare.

La apel cuvintele cheie au în faţă caracterul ":"

Aventajul (deşi sunt opţionale) e că pot apărea în orice ordine sau poziţie la apel.

```
> (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)

;; un alt exemplu

> (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 Liste de asociere

Elementele unei liste de asociaţie sunt celule cons în care parţile aflate în car se numesc chei şi cele aflate în cdr se numesc date. Pentru a introduce şi pentru a extrage noi elemente, de regula, operăm asupra unui capăt al listei, comportamentul este analog stivelor. Într-o astfel de structură, introducerea unei noi perechi cheie-dată cu o cheie identică uneia deja existentă are semnificaţia "umbririi" asociaţiei vechi, după cum eliminarea ei poate sa însemne revenirea "în istorie" la asociaţia anterioară. Pe acest comportament se bazează, de exemplu, "legarea" variabilelor la valori.

```
(setq lucruri '((telefon buzunar)
                (bani card)
                (caiet ghiozdan)
                (nota carnet_student)))

> (assoc 'caiet lucruri)

> (assoc 'nota lucruri)

> (assoc 'mapa lucruri)

;; adaugarea unei noi asocieri

> (setq lucruri (cons '(geaca dulap) lucruri))

> (assoc 'geaca lucruri)

;; pentru a economisi spatiu putem folosi perechi cu punct

(setq lucruri '((telefon . buzunar)
                (bani . card)
                (caiet . ghiozdan)
        (bani . buzunar2)
                (nota . carnet_student)))

> (assoc 'bani lucruri)

;; pentru a gasi bani o solutie ar fi sa folosim:

(setq lucruri '((telefon . buzunar)
                (bani_euro . card)
                (caiet . ghiozdan)
        (bani_ron . buzunar2)
                (nota . carnet_student)))

> (assoc 'bani_euro lucruri)

> (assoc 'bani_ron lucruri)
```

```
> (rassoc 'card lucruri)   ;;; functioneaza doar pentru listele cu punct
```

Funcţiile de acces într-o listă de asociere sunt:

- *assoc* pentru acces la cheia elementului cautat;
- *rassoc* pentru acces la data;

Încă un exemplu:

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

> valori

> (assoc ' a valori)

> (rassoc 1 valori)
```

Funcţia *pairlis* organizează o listă de asociaţie din chei aflate într-o listă şi date aflate în alta. Nu există o ordine a-priorică de introducere a elementelor în lista de asociaţie. Evident, cele doua liste-argument trebuie sa aibă aceeasi lungime.

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

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

Locaţii şi acces la locaţii:

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

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

> lista
```

4.1 Problema

Scrieţi o funcţie ce returnează o listă cu toate cheile prezente într-o listă de asociere.

4.2 Problema

Scrieţi o funcţie assoc-all ce returnează sublistele tuturor aparţiilor unei anumite chei într-o listă de asociere.

4.3 Liste de proprietăţi p-lists

Lista de proprietăţi a unui simbol:
Unui simbol i se poate asocia o listă de perechi proprietate-valoare, în care proprietăţile sunt simboluri, iar valorile sunt date Lisp. În această listă o proprietate poate să apară o singură dată. O listă de proprietăţi are asemănări cu o listă de asociaţie (astfel numele de proprietate corespunde cheii, iar valoarea proprietăţii corespunde datei) dar exista si diferenţe între ele (în lista de proprietăţi

o singura valoare poate fi atribuită unei proprietăţi, dar mai multe în lista de asociaţie, ce pot fi regăsite în ordinea inversă a atribuirilor). Implementaţional, o listă de proprietăţi a unui simbol este o listă de lungime pară, în care pe poziţiile impare sunt memorate (cu unicitate) numele proprietăţilor si alături de ele, pe poziţiile pare valorile acestora. Cercetarea valorii unei proprietăţi a unui simbol se face prin funcţia *get*:

Exemplu1: (valoare-funcţie)

```
> (setq s 10)

> s

> (symbol-value 's)

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

> (s 9)

> (symbol-function 's)

> (symbol-value 's)
```

Exemplu2: Lista de proprietăţi ale unui simbol:

```
> (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)
```

Alt exemplu:

```
> (setq program nil)

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

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

> (setf (get 'program 'varianta) '("CLisp" "Quintus Prolog"))

> (get 'program 'varianta)

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

> (get 'program 'versiune)

> program

> (get 'program 'sistem_operare)

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

> (get 'sky 'colour)

> (remprop 'sky 'colour)

> (get 'sky 'colour)
```

4.4 Exerciţiu - Data abstraction

Sau cum sa nu ne pierdem in detalii de implementare...

In anumite situaţii poate fi convenabilă reprezentarea datelor sub forma valorilor simple, a listelor de asociere sau a listelor de proprietati.

Programatorii experimentati însă, obişnuiesc sa gandeasca in alti termeni, si anume, se izoleaza de aceste detalii de nivel jos si se concentreaza asupra conceptelor de nivel mai inalt - asa cum, de pilda, un arhitect nu dedica cea mai mare parte a timpului sau alegerii caramizilor, ci priveste cladirea in termeni mai generali de forte, echilibru, structura de sustinere, spatii cu destinatii specifice etc.

In programare, acest tip de gandire este sustinut de realitatea ca este usor sa scriem proceduri pentru crearea, accesarea si actualizarea datelor - data constructors, data selectors, data mutators sau access and update procedures.

Pentru a exemplifica acest mod de a gandi, sa consideram ca vrem sa descriem un castel de jucarie, facut din cuburi (paralelipipede) si piramide. Fiecare

cub are anumite caracteristici stocate sub forma unei liste de asociere. Piramidele au si ele caracteristici stocate sub forma unor liste de proprietati.

Pentru un cub ales la întâmplare, numit cub-b, caracteristicile ar putea fi reprezentate astfel:

```
((este cub) (culoare rosu) (sustinut-de cub-a) (material lemn))
```

Pentru a modifica aceste caracteristici, vom proceda astfel:

```
(setf (get 'cub-b 'a-list) '((este cub) (culoare rosu)
(sustinut-de cub-a) (material lemn)))
```

Pentru a accesa caracteristicile, va trebui deci sa procedam astfel:

```
(cadr (assoc 'culoare (get 'cub-b 'a-list))) => rosu
```

Sa presupunem ca aceste caracteristici sunt stocate într-o proprietate a cuburilor numita a-list. Pentru piramide, caracteristicile ar putea fi, de pilda, stocate chiar ca valori ale unor proprietati.

```
(get 'piramida-c 'este) => piramida
(get 'piramida-c 'culoare) => rosu
(get 'piramida-c 'material) => plastic
```

In concluzie, va trebui sa retinem ca pentru cuburi vom folosi

```
(cadr (assoc '<nume-caracteristica> (get '<nume-cub> 'a-list)))
```

iar pentru piramide vom folosi

```
(get '<nume-piramida> '<nume-caracteristica>).
```

Evident, ar fi total incomod sa avem in permanenta in vedere aceste detalii de nivel jos, pe langa problema propriu-zisa a modelarii, asa ca ne vom defini selectori pentru fiecare tip de data astfel:

```
(defun culoare-cub (cub)
  (cadr (assoc 'culoare (get cub 'a-list))))

(defun culoare-piramida (piramida)
  (get piramida 'culoare))

; Constructorii ii putem defini astfel:

(defun cons-cub (obiect culoare material sustinut-de)
  (setf (get obiect 'a-list)
    (list (list 'este 'cub)
          (list 'culoare culoare)
          (list 'material material)
          (list 'sustinut-de sustinut-de)))
  obiect)

(defun cons-piramida (obiect culoare material sustinut-de)
  (setf (get obiect 'este) 'piramida)
  (setf (get obiect 'culoare) culoare)
  (setf (get obiect 'material) material)
  (setf (get obiect 'sustinut-de) sustinut-de)
```

```
  obiect)

; Pentru a specifica caracteristicile obiectelor vom scrie
; functii de actualizare care arata in acest caz astfel:

(defun modi-culoare-cub (cub culoare)
  (setf (get cub 'a-list) (cons (list culoare) (get cub 'a-list)))
  culoare)

(defun modi-culoare-piramida (piramida culoare)
  (setf (get piramida 'culoare) culoare))

; Cand avem selectori si modificatori ce se aplica la doua
; sau mai multe tipuri de date, spunem ca avem de-a face cu
; un tip de date compus. In acest caz ar fi mai inteligent sa
;  scriem selectori si modificatori care sa se ocupe toate
; tipurile de date.
; In cazul nostru, tipul de date compus ar fi cel numit "bloc"
; si vom scrie un selector pentru el astfel:

(defun culoare-bloc (bloc)
  (cond ((equal (get bloc 'este) 'piramida)
         (culoare-piramida bloc))
        (T (culoare-cub bloc))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;; test

> (cons-cub 'cub-b 'rosu 'lemn 'cub-a)

> (culoare-cub 'cub-b)

> (cons-piramida 'piramida-c 'albastru 'plastic 'cub-d)

> (culoare-piramida 'piramida-c)

> (culoare-bloc 'piramida-c)

> (culoare-bloc 'cub-b)


> (modi-culoare-cub 'cub-b 'verde)

> (culoare-bloc 'cub-b)
```

5 Temă

Construiţi o listă de asociere casă. Această casă conţine mai multe camere şi în fiecare cameră se află cel puţin un obiect. Exemplu: (hol cuier) (sufragerie masa scaun1 scaun2 televizor canapea) ş.a.m.d.

Adăugaţi şi ştergeţi proprietăţi ale obiectelor. De exemplu: masa din sufragerie este din lemn si are o anumită dimensiune.

Scrieţi o funcţie planulcasei care afişează planul casei. Exemplu: "Ne aflăm în casa. La început intrăm în hol, lăsăm haina în cuier, apoi intrăm în sufragerie unde se află o masă de lemn care are dimensiunea ..., scaun1, scaun2, ..." ş.a.m.d.