Programare Funcţională – Laborator 5

Definire de noi funcţii

Isabela Drămnesc

Martie 2023

1 Concepte

- Iterativitate
- remove, remove*
- expresii lambda
- apply
- map, andmap, ormap, for-each, foldl
- Legare statică şi legare dinamică
- Închideri

2 Discutarea temei din laboratorul 4

3 Exerciţii

3.1 Simulaţi comportamentul interpretorului Racket pentru fiecare din urmatoarele:

```
1)
(do ((x 0 (+ 1 x))
     (y 0 (+ x y))
     (z 0 (+ y z)))
   ((>= x 10)
  (list x y z))
  (print (list x y z)))

2)
(do ((v1  1 (+ 1 v1))
        (v2 () (cons 'a v2)))
        ((> v1 5)
                v2)
        (print v1))
```

```
3)
(do ((v1 1 (print (+ 1 v1)))
     (v2 (print ()) (cons 'a v2)))
    ((> v1 0)   v2)
   (print (list v1 v2)))


4)
(do ((v1 1 (+ 1 v1))
     (v2 'nothing (cons 'a v2)))
        ((> v1 0)  v2)
        (print v1))


5)
(do ((v1  1 (+ 1 v1))
     (v2 () ))
       ((> v1 3) v2)
       (set! v2 (cons 'a v2)))



6)
(do ((v1  1 (+ 1 v1))
     (v2 () (cons 'a v2)))
        ((> v1 3) v2)
        (set! v2 (cons 'b v2)))


7)
(do ((v1   1  (+ 1 v1))
     (v2 '(b) (cons 'a v2))
     (v3   1  (list v1 v2 v3)))
     ((> v1 3) v2)
     (print (list v1 v2 v3)))


8)
(do ((v1 1 (+ 1 v1))
     (v2 () (cons 'a v2))
     (v3 9))
        ((print v1)  ; end cond
        )                    ; no return value
        (print v3))
```

3.2 remove, remove*

```
> (define l '(azi e soare))
```

```
> (remove 'soare l)

> l

> (remove* '(-1) '(3 -1 3 -1 0 -1 2 4))

> (remove '(-1) '(3 -1 3 -1 0 -1 2 4))
```

3.3 Expresii lambda

Apar:

- când o funcţie e folosită o singură dată şi e mult prea simplă ca să merite a fi definită;
- funcţia de aplicat trebuie sintetizată dinamic (fiind imposibil să fie definită cu DEFINE).

Sintaxă:
(lambda l f1 f2 f3 ... fn)
sau
((lambda l f1 f2 f3 ... fn) par1 par2 ... parn)

- defineşte o funcţie utilizată local;
- l reprezintă lista parametrilor; (poate fi dată explicit (număr fix de parametri) sau parametric (lista l având un număr variabil de parametri);
- f1 f2 ... fn corpul funţiei;

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
)

; se poate scrie mai simplu (exemplul urmator)
```

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

Exemplu:

```
(define aduna_cu_3
        (lambda (x) (+ x 3)))

> (aduna_cu_3 10)

>(aduna_cu_3 26.6)
```

3.4 Apply

Există cazuri în care numărul parametrilor unei funcţii trebuie stabilit dinamic. Aplicarea unei funcţii asupra unei mulţimi de parametri sintetizată eventual dinamic este posibilă cu ajutorul funcţiei APPLY.

Sintaxă:
(apply functie (listaparametri))

Exemple:

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

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

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

> (apply + 1 2 '(10))

> (apply + 1 2 '(10 20))
```

Definiţi o funcţie (ca o variantă a funcţiei apply) care permite aplicarea unei funcţii la un număr fix de parametri.

```
(define (funcall fun . args)
        (apply fun args))

> (funcall + 1 2 3 4)

> (apply + 1 2 3 4)
ERROR

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

> (funcall (lambda (a b) (+ a b)) 2 3)
```

```
> (funcall newline)

> (apply newline)

> (apply newline '())
```

Exemple:

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

> (funcall max 1 2 3 4 5 6)

> (define p 'car)

> ((eval p) '(a b c))

> (funcall (eval p) 'a 'b 'c))

> (apply (eval p) '((a b c)))

> (define f1 (lambda(x) (+ x 3)))

> (funcall f1 5)

> f1
```

3.5 Map, andmap, ormap, for-each, foldl

Map este o funcţie de aplicare globală. Ea se aplică pe rând asupra elementelor listei argument, rezultatele fiind culese într-o listă întoarsă ca valoare a apelului funcţiei MAP. Evaluarea se încheie la terminarea listei celei mai scurte.

Sintaxă:
(map proc lst ...+) ? list?
proc : procedure?
lst : list?

```
> (map + '(1 2 3) '(1 2 3))

> (map + '(1 2) '(1 2 3))
ERROR

> (map + '(1 2 3) '(1 2))
ERROR

> (map + '(1 2 3) '(1 2 3) '(1 2 3))

> (map (lambda (number) (+ 1 number)) '(1 2 3 4))

> (map (lambda (number1 number2) (+ number1 number2))
          '(1 2 3 4)
          '(10 100 1000 10000))
```

```
> (andmap positive? '(1 2 3))

> (andmap positive? '(1 2 a))

> (andmap positive? '(1 -2 a))

> (andmap positive? '(1 a -2))

> (andmap + '(1 2 3) '(4 5 6))

> (andmap + '(1 2 3) '(4 5 6 10 20))

> (andmap odd? '(1 2 3))

> (andmap pair? '(1 2 3))

> (map cons '(1 2 3) '(4 5 6))

> (map car '((a b c) (x y z)))

> (map car '((a b c)))

> (ormap eq? '(a b c) '(a b c))

> (ormap positive? '(1 2 a))

> (ormap + '(1 2 3) '(4 5 6))

> (for-each (lambda (arg)
        (printf "Elementul ~a\n" arg)
        23)
        '(1 2 3 4))

> (for-each
        (printf "Elementul \n")
        '(1 2 3 4))
        ERROR

> (foldl cons '() '(1 2 3 4))

> (foldl + 0 '(1 2 3 4))

> (foldl + 2 '(1 2 3 4))

> (foldl cons '(a) '(1 2 3 4))

> (foldl cons '(a b)
       '(1 2 3 4))
```

```
> (foldl (lambda (a b result)
         (* result (- a b)))
         1
         '(1 2 3)
         '(4 5 6))
```

3.6 Liste circulare

Studiaţi următorul exemplu:

```
(define (list-len x)
  (do ((n 0 (+ n 2))                ;Contor
       (fast x (cddr fast))          ;Pointer rapid: merge din 2 in 2
       (slow x (cdr slow)))          ;Pointer incet: trece prin fiecare cdr
       (#f)
       ;; Daca pointerul rapid atinge sfarsitul intoarce n.
       (when (null? fast) n)
       ;; Daca cdr-ul pointerului rapid atinge este cel final, intoarce n+1.
       (when (null? (cdr fast)) (+ n 1))
       ;; Daca pointerul rapid il ajunge din urma pe cel incet,
       ;; inseamna ca avem de-a face cu o lista circulara.
       (when (and (eq fast slow) (> n 0)) '())))
```

Reparaţi dacă e necesar!

3.7 Propriul nostru predicat de egalitate

Functie care decide "egalitatea" a doua structuri formate cu cons-uri:

```
(define (egal l1 l2)
  (cond ((and (null? l1) (null? l2)) #t)
        ((and (or (symbol? l1) (number? l1))
        (or (symbol? l2) (number? l2))) (equal? l1 l2))
        ((and (or (symbol? l1) (number? l1))
        (not (or (symbol? l2) (number? l2)))) #f)
        ((and (not (or (symbol? l1) (number? l1)))
        (or (symbol? l1) (number? l1))) #f)
        ((egal (car l1) (car l2)) (egal (cdr l1) (cdr l2)))
        (#t #t)
   )
)
```

4 TEMA

4.1 RANGE şi VALID-RANGE

Definiţi o funcţie iterativă RANGE care primeşte ca şi parametru o listă de numere şi returnează o listă de lungime 2 care conţine cel mai mic număr şi cel mai mare număr. Utilizaţi predicate > şi < Asiguraţi-vă că lista e parcursă

o dată. Scrieţi încă o funcţie VALID-RANGE, care returnează acelaşi rezultat ca RANGE dacă elementele listei sunt toate numere şi dacă nu returnează INVALID.
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)
```

4.2 Scrieţi 2 expresii pentru a accesa simbolul C pentru fiecare din listele următoare:

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

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

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

4.3 Cum schimbaţi C în SEE pentru fiecare din listele următoare:

```
(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)
```

4.4 Problemele din cursul 4 (ultimul slide) în variantele:

1. recursivă
2. final recursivă
3. iterativă.

Notă: Termen de realizare: laboratorul următor.