Programare funcţională – Laboratorul 2

Definirea de noi funcţii

Isabela Drămnesc

March 4, 2014

1 Concepte

- Variabile locale, globale, constante
- Atribuire, Egalitate
- If, Cond
- Definire de noi funcţii

2 Intrebari din laboratorul 1

- Care este diferenţa dintre cons, list, append?
- Cum se reprezintă listele? Exemplificaţi pentru lista (azi (e 5) martie)
- Ce va returna: > (cdar '((a (b c)) d ((e f) g) h))

3 Atribuire, Variabile locale, globale

(let ((var expr) ...) body1 body2 ...)

```
>(let ((x 2) (y 3)) (+ x y))

> x

> y

> (let ([f +]) (f 10 20))

> (let ([+ *])
  (+ 20 30))

> (let ([x 1])
  (let ([x (+ x 1)])
    (+ x x)))

; variabilele definite cu let sunt vizibile doar in interiorul lui let
```

```
> (let ([x 1])
  (let ([new-x (+ x 1)])
    (+ new-x new-x)))
```

Determinati valoarea urmatoarei expresii. Explicati cum a fost obtinuta aceasta valoare. Redenumiti variabilele in asa fel incat sa evidentiati mai bine obtinerea valorilor.

```
(let ([x 9])
  (* x
     (let ([x (/ x 3)])
       (+ x x))))

(let ([x 'a] [y 'b])
  (list (let ([x 'c]) (cons x y))
        (let ([y 'd]) (cons x y))))
```

DEFINE:
(define (var0 var1 ... varn) e1 e2 ...)

```
(define abc '(a b c))

> abc

(define abcde '(a b c d e))

> abcde

(set! abcde (cdr abcde))

(let ([abcde '(a b c d e)])
  (set! abcde (reverse abcde))
   abcde)

> abcde

(let ([d 0]) (set! d '(a b c)) d)

> d

(define d '(a b c))

> d
```

procedure: (set-car! pair obj)
set-car! seteaza car-ul lui pair in obj.

```
(let ([x (list 'a 'b 'c)])
  (set-car! x 1)
  x)
```

(set-cdr! pair obj)
set-cdr! seteaza cdr-ul lui pair in obj.

```
(let ([x (list 'a 'b 'c)])
  (set-cdr! x 1)
  x)
```

(let* ((var expr) ...) body1 body2 ... bodyn) returneaza: valorile evaluarii expresiei bodyn

let* e similar cu let diferenta e ca expresiile expr ... sunt evaluate secvential de la stanga la dreapta, si fiecare dintre ele este atribuita uneia din variabilele var

```
(let* ([x (* 5.0 5.0)]
       [y (- x (* 4.0 4.0))])
  (sqrt y))

(let ([x 0] [y 1])
  (let* ([x y] [y x])
    (list x y)))
```

4 Egalitatea - o chestiune netriviala in Racket

```
; (eq? obj1 obj2)
; returneaza: true daca obj1 si obj2 sunt identice, false altfel

; EQ este egalitatea cea mai puternica.

> (eq? 'a 3)

> (eq? #t 't)

> (eq? "abc" 'abc)

> (eq? "hello" '(hello))

> (eq? #f '())

> (eq? 9/2 4.5)

> (eq? 3 3.)

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

> (eq? 9/2 9/2)

> (let ([x (* 12345678987654321 2)])
  (eq? x x))

> (eq? #\a #\b)
```

```
> (eq? #\a #\a)

> (let ([x (string-ref "hi" 0)])
  (eq? x x))

> (eq? #t #t)

> (eq? #f #f)

> (eq? #t #f)

; Predicatul null? returneaza true daca argumentul sau este
; lista vida () si false altfel.

> (null? '())

> (null? 'abc)

> (null? '(x y z))

> (null? (cdddr '(x y z)))

> (eq? (null? '()) #t)

> (eq? (null? '(a)) #f)

> (eq? (cdr '(a)) '())

> (eq? 'a 'a)

> (eq? 'a 'b)

> (eq? 'a (string->symbol "a"))

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

> (eq? '(a) '(a))

> (let ([x '(a . b)]) (eq? x x))
```

(eqv? obj1 obj2) returneaza: true daca obj1 si obj2 sunt echivalente, false altfel

eqv? e similar cu eq? cu diferentele ca eqv? returneaza true pentru doua caractere care sunt considerate egale prin char=? si pentru doua numere care (a) sunt considerate egale prin = si (b) care nu pot fi deosebite prin alte operatii in afara de eq? si eqv?. O consecinta a punctului (b) este ca (eqv? -0.0 +0.0) este false desi (= -0.0 +0.0) este true in sisteme care fac diferenta intre -0.0 si +0.0, (precum cele bazate pe aritmetica IEEE floating-point).

Similar desi 3.0 si 3.0+0.0i sunt considerate numeric egale, ele nu sunt in relatia eqv? deoarece au representari diferite.

```
(= 3.0+0.0i 3.0)

(eqv? 3.0+0.0i 3.0)

(eqv? #t #t)

(eqv? #f #f)

(eqv? #t #f)

(eqv? (null? '()) #t)

(eqv? (null? '(a)) #f)

(eqv? (cdr '(a)) '())

(eqv? 'a 'a)

(eqv? 'a 'b)

(eqv? 'a (string->symbol "a"))

(eqv? "abc" "cba")

(eqv? "abc" "abc")
```

(equal? obj1 obj2)

returneaza: true daca obj1 si obj2 au aceeasi structura si acelasi continut, false altfel

Doua obiecte sunt in relatia equal daca sunt in relatia eqv?, string-uri care sunt in relatia string=?, perechi ale caror car si cdr sunt egale, etc.

```
(equal? 'a 3)

(equal? #t 't)

(equal? "abc" 'abc)

(equal? "hi" '(hi))

(equal? #f '())

(equal? 9/2 7/2)

(equal? 3.4 53344)

(equal? 3 3.0)

(equal? 1/3 #i1/3)
```

```
(equal? 9/2 9/2)

(equal? 3.4 (+ 3.0 .4))

(let ([x (* 12345678987654321 2)])
  (equal? x x))

(equal? #\a #\b)

(equal? #\a #\a)

(let ([x (string-ref "hi" 0)])
  (equal? x x))

(equal? #t #t)

(equal? #f #f)

(equal? #t #f)

(equal? (null? '()) #t)

(equal? (null? '(a)) #f)

(let ([x '(a . b)]) (equal? x x))

(let ([x (cons 'a 'b)])
  (equal? x x))

(equal? (cons 'a 'b) (cons 'a 'b))

(equal? car car)
```

Concluzii:

- EQ intoarce T daca argumentele au ca valoare un acelasi obiect si false in caz contrar.
- EQL: doua elemente sunt EQL daca sunt EQ sau daca sunt numere intregi avand acelasi tip;
- EQUAL intoarce T daca valorile argumentelor sunt expresii echivalente (adica daca cele doua S-expresii au aceeasi structura)

5 Predicate cu mai multe argumente

```
> (> 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 se foloseste astfel:
(if <test> <expresie-then> [<expresie-else>])

```
(if #t 'true 'false)

(if #f 'true 'false)

(if '() 'true 'false)

(if 1 'true 'false)

(if '(a b c) 'true 'false)

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

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

> (if (+ 2 3) 1 2)


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


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

cond se foloseste astfel:

```
(cond
    (<test_1> <consecinta_1_1> <consecinta_1_2> ...)
    (<test_2>)
    (<test_3> <consecinta_3_1> ...)
    ...
)
```

Din punctul de vedere al lui if sau cond, orice expresie <test-n> care nu este false este acceptata ca true, chiar daca (equal <test-n> #t) este false.

```
> (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))
```

Întrebăm aşa:

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

Dacă x = true? (atunci returnează b)

Dacă x = false, y = true? (atunci returează c)

Dacă x = false, y = true? (atunci returnează d)

Alt exemplu:

```
(let ([x 0] [y 0] [z 0])
(cond (x (set! x 1) (+ x 2))
      (y (set! y 2) (+ y 2))
      (#t (set! x 0) (set! y 0))
))
```

Dacă x=true (atunci returnează 3) Cine e x?($x = 1$)

Dacă x=false, y=true (atunci returnează 4) Cine sunt x şi y? (false şi 2)

Dacă x=false, y=false (atunci returnează 0) Cine sunt x şi y? (0 şi 0)

Explicati urmatorul cod:

```
(let ([x 0] [y 0] [z 0])
(cond ((< x 0) (set! x 1) (+ x 2))
      ((< y 0) (set! y 2) (+ y 2))
      (#t (set! x 0) (set! y 0) (+ x y))
))
```

7 Funcţii utilizator

7.1 Definirea funcţiilor

(define (<nume-func> <lista-param>)
(<expr-1> <expr-2> ... <expr-n>))
unde:
<nume-func> este primul argument si reprezinta numele functiei definite de defun;

<expr-i>, i = 1, . . . , n sunt forme ce alcatuiesc corpul functiei definite.

```
(define f1 (+ 2 3))

> f1

(define (f2 vara varb)
   (+ vara varb))

> (f2 3 4)
(define (sum-of-squares x y)
  (+ (* x x) (* y y)))

> (sum-of-squares 10 20)

(define (ec-grad-2 a b c)
    (let ([root1 0] [root2 0] [minusb 0] [radical 0] [divisor 0])
      (set! minusb (- 0 b))
      (set! radical (sqrt (- (* b b) (* 4 (* a c)))))
      (set! divisor (* 2 a))
      (set! root1 (/ (+ minusb radical) divisor))
      (set! root2 (/ (- minusb radical) divisor))
      (cons root1 root2)))

> (ec-grad-2 2 -4 -6)

; sau o alta versiune

(define (ec-gr-2 a b c)
    (let ([minusb (- 0 b)]
          [radical (sqrt (- (* b b) (* 4 (* a c))))]
          [divisor (* 2 a)])
      (let ([root1 (/ (+ minusb radical) divisor)]
            [root2 (/ (- minusb radical) divisor)])
        (cons root1 root2))))

> (ec-gr-2 2 -4 -6)

(define (e-numar-par x)
             (equal? (modulo x 2) 0))

> (e-numar-par 6)

> (e-numar-par 7)

(define (abs n)
  (if (< n 0)
      (- 0 n)
      n
```

```
    )
  )
```

Scrieti o functie care are urmatorul efect?

```
> (numar-par-sau-divizibil-cu-7 7)
true

> (numar-par-sau-divizibil-cu-7 0)
true

> (numar-par-sau-divizibil-cu-7 10)
true

> (numar-par-sau-divizibil-cu-7 11)
false

> (numar-par-sau-divizibil-cu-7 14)
true

> (define (al-treilea lista)
          (car (cdr (cdr lista))))

> (al-treilea '(r t y))

> (al-treilea '(r t y h))


>(length '(c b a))

; Definim functia noastra "len" care returneaza lungimea unei liste

>(len '(c b a))

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

>(len '(1 2 3 4))

;;; nu avem caz de baza: ciclu infinit

;;; definim len intr-un mod corect -- recursiv

>(define (len list)
  (if (null? list)
        0                          ; base case
        (+ 1 (len (cdr list)))))  ; inductive case

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

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

>(len ())

>(len '(a))

>(len '(b a))

>(len '(c b a))

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

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

; O functie care ia ca si parametri doua liste
;returneaza true daca prima lista are lungimea mai mare
;decat a doua lista

> (define (longer-listp list1 list2)
(if
          (> (length list1) (length list2))
           #t
           #f
       )
)

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

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

8 Tema

Problema 1

Scrieti o functie care returneaza x la puterea y.

Problema 2

Scrieţi o funcţie care returneaza $x * x * y * y$.

Problema 3

Scrieţi o funcţie care returnează numărul de numere care apar într-o listă.

Problema 4

Scrieţi o funţie ACELEASI-ELEMENTE care primeste ca si argument o lista si care returneaza true daca toate elementele din lista sunt egale si false altfel.

Problema 5

Scrieţi o funcţie care primeşte ca parametri doua liste si returnează o nouă listă formată din concatenarea celor două liste.

Problema 6

Scrieţi o funcţie care calculează factorialul unui număr natural.

Notă: Termen de realizare: laboratorul următor.