

Racket: Introducere



PP

- programare procedurala: C
- programare orientata obiect: Java
- programare functionala: Racket, Haskell
- programare logica: Prolog

Efecte laterale

- prezente în programarea procedurală/orientată obiect
- nu întoarcem numai un rezultat, modificăm și starea unor entități - variabile, structuri, etc

Programare functionala = fara efecte laterale

- programare atemporală
- fara atribuiriri, nu exista secvențe de comenzi, etc
- programele = compuneri + aplicari de funcții

Asemanare cu axiomele unui TDA

→ TDA = tip de date abstract (de la AA)

Exemplu 0

- **sum lista:** calculeaza suma elementelor din lista
- $\text{sum } [] = 0$
- $\text{sum } (x:xs) = x + (\text{sum } xs)$

Exemplu 1

- **insert element list:** adauga elementul in lista pastrand lista sortata crescator
- **insert y [] = y**
- **insert y (x:xs) = if y < x then y:(x:xs) if else x:(insert y xs)**

Exemplu 2

- **insertion_sort lista: sorteaza lista folosind algoritmul insertion sort**
- **insertion_sort [] = []**
- **insertion_sort (x:xs) = insert x (insertion_sort xs)**

Wishful thinking

- putem folosi alte functii inainte sa le definim
- ex: pot folosi insert la insertion_sort si sa o definesc dupa

Racket 0

- derivat din Lisp
- tipat **dinamic** - nu verificam tipul valorilor la compilare
- tipat **strong** - nu se face conversie automata la un anumit tip

Racket 1

- cu **evaluare aplicativa** = argumentele functiilor sunt evaluate inainte de aplicarea functiei
- multiparadigma (suporta si unele componente imperative)

Tipuri de evaluari

- aplicativa (evaluam argumentele inainte de aplicarea functiei)
- normala (evaluam argumentele odata ce avem nevoie de ele)

Exemplu evaluare aplicativa (Racket)

- (define (first l) (car l))
- (first (append '(1 2) '(3 4))) = (first '(1 2 3 4)) = (car '(1 2 3 4)) = 1

Exemplu evaluare normală (Haskell)

→ `first l = (head l)`

→ `first ([1, 2] ++ [3, 4]) = head ([1, 2] ++ [3, 4]) = head [1, 2, 3, 4] = 1`

Apeluri functii vs liste

- paranteza deschisa => urmeaza un apel de functie
- daca vrem o lista => ' inainte de paranteza

Exemplu: (= 1 2)

- = este o functie
- aplicata pe cei doi parametri: 1, 2
- intoarce false pentru $1 \neq 2$

Tipuri de date simple

- true(#t), false(#f)
- numar(ex: 1, 2, etc)
- simbol(ex: 'a, 'abc, etc)

Tipuri de date compuse

- pereche(ex: '(1 2)')
- lista(ex: '(1 2 3 4)')
- string(ex: "ceva")
- vectori

Simbol (literali): 'simbol

- 1+ caractere fara spatii
- ' evita evaluarea simbolului
- daca lipseste ' => se cauta variabila cu numele simbolului

Perechi



cons a b

- creeaza o pereche cu a first si b second (. intre ele, altfel e lista)
- (cons 1 2) => '(1 . 2)
- (cons '(1) 2) => '((1) . 2)
- daa, (cons '(1) '(2)) => '((1) 2) // lista, nu pereche

car (primul element din pereche)

- (car '()) => eroare
- (car '(1 . 2)) => 1
- (car '((1) 2)) => '(1)

cdr (al doilea element din pereche)

- (cdr '()) => eroare
- (cdr '(1 . 2)) => 2
- (cdr '(1 . (2))) => '(2)

Liste

- pot să aiba elemente cu tip diferit: (list 1 'a 5 100)
- perechi generalizate (pereche dintre primul element si restul listei)

cons a b = adauga elementul a in lista b

- (cons 1 null) => '(1)
- (cons 1 '(2 3)) => '(1 2 3)
- (cons '(1 2) '(3)) => '((1 2) 3)

cons a b - liste vs perechi

- dacă b este element => rezultă o pereche
- dacă b este lista => rezultă o listă

list ... = creeaza o lista din argumente

- (list null) => '()
- (list 1 2 (+ 2 4)) => '(1 2 6) // mai intai se evaluateaza (+ 2 4) dupa se face lista
- daa, '(1 2 (+ 2 4)) => '(1 2 (+ 2 4)) // daca e apostrof in fata nu se mai evaluateaza dupa
- apostrof = un fel de escape

car, cdr, etc 0

- (car '(1 2 3)) => 1
- (cdr '(1 2 3)) => '(2 3)
- (cdr '(1 2)) => '(2) // la liste cdr intoarce mereu o lista

car, cdr, etc 1

→ (car (cdr '(1 2 3))) => 2

→ (cdr (car (cdr '(1 (2 3))))) => '(3)

→ (cdaddr '(1 (2 3))) => '(3) // ultimele 2 sunt echivalente

**take/drop n l = intoarce/scoate primele n
elemente din l**

- (take '(1 2 3) 2) => '(1 2)
- (take '(1 2 3) 20) // eroare (trebuie cel putin 20 elemente)
- (drop '(1 2 3) 2) => '(3)
- (drop '(1 2 3) 20) // eroare

take-right/drop-right n |

- asemanător cu take/drop, dar elementele se extrag acum din dreapta
- (take-right '(1 2 3) 2) => '(2 3)
- (drop-right '(1 2 3) 2) => '(1)
- (take-right '(1 2 3) 20), (drop-right '(1 2 3) 20) // eroare

Alte functii pe liste 0

- (append l1 l2 l3 ...) poate primi **oricate** liste
- (null? l)
- (length l)

Alte functii pe liste 1

→ (reverse l)

→ (list? l)

define

- folosit pentru definirea variabilelor, functiilor, etc
- nu se poate defini acelasi simbol de mai multe ori

Exemple define 0

- (define a 1) // nu intoarce nimic
- a => 1 // cauta identificatorul cu numele a si il afiseaza
- (define b a) // evaluateaza a si salveaza b = 1
- b => 1

Exemplu define 1

- (define c 'a) // apostrof => a nu se evaluateaza
- c => 'a
- (define d (+ 1 (+ (max 1 10) (min -8 10)))) // se evaluateaza pe rand expresiile
- d => 3

Functii lambda

- (lambda (lista argumente) valoare intoarsa)
- ex: (lambda (x y) (max x y))
- ex aplicare functie lambda: ((lambda (x y) (max x y)) 1 10)

Functii lambda + define

- (define maxi (lambda (x y) (max x y)))
- (maxi (maxi 1 80) -10) // 80
- (define comp (lambda (f g x)(f (g x))))
- (comp list list 1) // '((1))

Functii utile

- aritmetice: +, -, *, /, modulo, quotient
- relationali: <, <=, >, >=, =, eq?, equal?
- logici: not, and, or, zero?, positive?, negative?, even?, odd?

Atentie la erori 0

- (or #t (> (/ 1 0) 2)) // #t
- orice valoare are (> (/ 1 0) 2), #t sau aia tot #t da
- **nu se mai evaluateaza (> (/ 1 0) 2) => nu se genereaza eroare**
- la fel, (and #f (> (/ 1 0) 2)) intoarce mereu #f si nu se va genera eroare

Atentie la erori 1

- dacă (and #t (> (/ 1 0) 2)) sau (or #f (> (/ 1 0) 2)) generează erori
- la ambele trebuie evaluată paranteza și acolo se generează și eroare

eq? vs equal?

- eq? a b: verifica daca a si b refera acelasi obiect
- equal? a b: verifica daca a si b au aceeasi valoare

Exemple eq?, equal?

- (eq? (list 1 2) (list 1 2)) // #f
- (eq? '(1 2) '(1 2)) // #f pentru ca sunt 2 liste diferite chiar daca s-a folosit '
- (equal? (list 1 2) (list 1 2)) // #t

(if testexpr return_true return_false) - exemple

→ (if (> a 0) a (* -1 a))

→ (if (= b 0) (if (> a 0) a (* -1 a)) (+ a (+ a 1)))

Alt exemplu if

- (if (= 12) (/ 1 0) (+ 1 2)) // 3
- mai intai se evalueaza conditia si in functie de valoarea ei(#t/#f), se evalueaza numai una dintre ramuri(then/else)

(cond (testexpr1 exp1) .. (else exp...)) -
exemplu

```
→ (cond  
    → ((= a 1) "1")  
    → ((= a 10) "10")  
    → (else "?"))
```

Alt exemplu cond

```
→ (define a 1)
→ (cond
  → ((< a 10) "digit") // se face match aici si se afiseaza "digit"
  → ((> a 0) "positive")
  → (else "no idea"))
```

cond = un fel de switch

- se opreste la prima conditie adevarata si evaluateaza rezultatul intors
- restul cazurilor nu se mai verifica

Comentarii

→ cu ; in fata



Exemple lab

- factorial
- suma elementelor dintr-o lista