Racket CheatSheet

Laborator 2

Recursivitate pe stivă

```
; suma elementelor unei liste
(define (sum-list L)
 aici nu avem nevoie de funcție auxiliară
  (if (null? L)
            ; la sfârșit creăm valoarea inițială
      (+ (car L) (sum-list (cdr L)))
      ; ^ construim rezultatul pe revenire
        (după întoarcerea din recursivitate)
 fiecare apel recursiv întoarce rezultatul corespunzător
    argumentelor
 concatenarea a două liste
(define (app L1 L2)
  (if (null? L1)
      L2 ; când L1 este vidă, întoarcem L2
       (cons (car L1) (app (cdr L1) L2))
       ; ^ construim rezultatul pe revenire))
   • fiecare apel recursiv se pune pe stivă
```

- complexitate spațială O(n)
- scriere mai simplă

Recursivitate pe coadă

```
; suma elementelor unei liste
 (define (sum-list L)
   (sum-list-tail 0 L)) ; <-- funcție ajutătoare
                       valoarea inițială pentru sumă
                     ; în sum construim rezultatul
 (define (sum-list-tail sum L)
   (if (null? L)
        sum
                     ; la sfârșit avem rezultatul gata
        (sum-list-tail
          (+ sum (car L))
             construim rezultatul pe avans
         ; (pe măsură ce intrăm în recursivitate)
           (cdr L))))
    ; funcția întoarce direct rezultatul apelului recursiv – toate
        apelurile recursive întorc același rezultat, pe cel final
   concatenarea a două liste
 (define (app A B)
   (app-iter B (reverse A)))
   ; nevoie de funcție ajutătoare
   ; rezultatul este construit în ordine inversă
 (define (app-iter B Result)
   (if (null? B); la sfârșit rezultatul e complet
        (reverse Result); inversăm rezultatul
        (app-iter (cdr B) (cons (car B) Result))))
           ; construim rezultatul pe avans
```

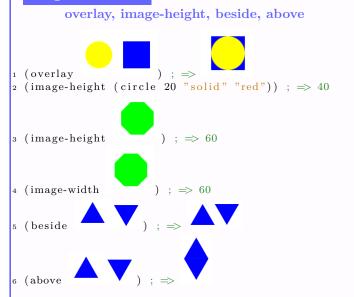
- apelurile recursive nu consumă spațiu pe stivă execuția este optimizată știind că rezultatul apelului recursiv este întors direct, fără operații suplimentare.
- complexitatea spațială este dată doar de spațiul necesar pentru acumulator de exemplu la sum-list-tail complexitatea spațială este O(1).
- scriere mai complexă, necesită de multe ori funcție auxiliară pentru a avea un parametru suplimentar pentru construcția rezultatului (rol de acumulator), mai ales dacă tipul natural de recursivitate al funcției este pe stivă.
 - Atenție: uneori, rolul acumulatorului poate fi preluat de unul dintre parametri, caz în care nu este nevoie nici de funcția suplimentară.
- rezultatul este construit în ordine inversă

Sintaxa Racket

AŞA DA / AŞA NU

```
1 DA: (cons x L) NU: (append (list x) L)
2 NU: (append (cons x '()) L)
3 DA: (if c vt vf) NU: (if (equal? c #t) vt vf)
4 DA: (null? L) NU: (= (length L) 0)
5 DA: (zero? x) NU: (equal? x 0)
6 DA: test NU: (if test #t #f)
7 DA: (or ceval ceva2) NU: (if ceval #t ceva2)
8 DA: (and ceval ceva2) NU: (if ceval ceva2 #f)
```

Imagini în Racket



Folosiți cu încredere!

http://docs.racket-lang.org/