

Racket CheatSheet

Laborator 2

Recursivitate pe stivă

```
1 ; suma elementelor unei liste
2 (define (sum-list L)
3
4
5 ; aici nu avem nevoie de funcție auxiliară
6
7
8 (if (null? L)
9     0 ; la sfârșit creăm valoarea inițială
10    (+ (car L) (sum-list (cdr L))))
11 ; ^ construim rezultatul pe revenire
12 ; (după întoarcerea din recursivitate)
13 ))
14 ; fiecare apel recursiv întoarce rezultatul corespunzător
    argumentelor
```

```
15
16
17 ; concatenarea a două liste
18
19 (define (app L1 L2)
```

```
20
21
22
23
24
25
26 (if (null? L1)
27     L2 ; când L1 este vidă, întoarcem L2
28     (cons (car L1) (app (cdr L1) L2)))
29 ; ^ construim rezultatul pe revenire))
```

- fiecare apel recursiv se pune pe stivă
- complexitate spațială $O(n)$
- scriere mai simplă

Recursivitate pe coadă

```
1 ; suma elementelor unei liste
2 (define (sum-list L)
3   (sum-list-tail 0 L)) ; <-- funcție ajutătoare
4                       ; ^ valoarea inițială pentru sumă
5
6                       ; în sum construim rezultatul
7 (define (sum-list-tail sum L)
8   (if (null? L)
9       sum ; la sfârșit avem rezultatul gata
10      (sum-list-tail
11        (+ sum (car L))
12        ; ^ construim rezultatul pe avans
13        ; (pe măsură ce intrăm în recursivitate)
14        (cdr L))))
15 ; funcția întoarce direct rezultatul apelului recursiv – toate
    apelurile recursive întorc același rezultat, pe cel final
```

```
16
17
18 ; concatenarea a două liste
19 (define (app A B)
20   (app-iter B (reverse A)))
21 ; nevoie de funcție ajutătoare
22 ; rezultatul este construit în ordine inversă
23
24 (define (app-iter B Result)
25   (if (null? B) ; la sfârșit rezultatul e complet
26       (reverse Result) ; inversăm rezultatul
27       (app-iter (cdr B) (cons (car B) Result))))
28 ; 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

(nume_functie arg1 arg2 ...)

```
1 (max 2 3)      3
2 (+ 2 3)        5
```

AȘA DA / AȘA 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 ceva1 ceva2)	NU: (if ceva1 #t ceva2)
8 DA: (and ceva1 ceva2)	NU: (if ceva1 ceva2 #f)

Imagini în Racket

overlay, image-height, beside, above

```
1 (overlay (image (circle 20 "solid" "red")) (image (square 20 "solid" "blue"))) ; =>
2 (image-height (circle 20 "solid" "red")) ; => 40
3 (image-height (image (circle 20 "solid" "red")) (image (square 20 "solid" "blue"))) ; => 60
4 (image-width (image (circle 20 "solid" "red")) (image (square 20 "solid" "blue"))) ; => 60
5 (beside (image (circle 20 "solid" "red")) (image (square 20 "solid" "blue"))) ; =>
6 (above (image (circle 20 "solid" "red")) (image (square 20 "solid" "blue"))) ; =>
```

Folosiți cu încredere!

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