Paradigme de programare

Laborator 2 - Racket: Recursivitate

Tipuri de recursivitate

Recursivitate pe stivă

O funcţie este **recursivă pe stivă** dacă apelul recursiv este parte a unei expresii mai complexe, fiind necesară **reţinerea** de informaţii, pe stivă, pe avansul în recursivitate.

Recursivitate pe coadă

O funcţie este **recursivă pe coadă** dacă valoarea întoarsă de apelul recursiv constituie valoarea de retur a apelului curent, i.e. apelul recursiv este un tail call, **nefiind necesară reţinerea** de informaţie pe stivă.

Exemplul 1

- **Calculul sumei elementelor dintr-o listă**

1. **Recursivitate pe stivă**

```
(define (sum L)
  (if (null? L)
    0 ;
    (+ (car L) (sum (cdr L)))))
```

2. **Recursivitate pe coadă**

```
(define (tail-recursion L acc)
  (if (null? L)
    acc
    (tail-recursion (cdr L) (+ acc
        (car L)))))

(define (sum L)
  (tail-recursion L 0))
```

Exemplul 2

- **Determinarea elementelor pare dintr-o listă**

1. **Recursivitate pe stivă**

```
(define (even-items L)
  (if (null? L)
    null
    (if (= (modulo (car L) 2) 0)
      (cons (car L) (even-items (cdr
          L)))
      (even-items (cdr L)))))
```

2. **Recursivitate pe coadă**

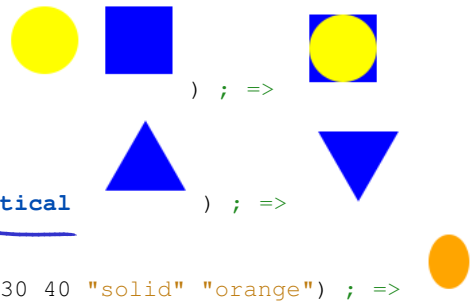
```
(define (even-items L)
  (tail-recursion L '()))

(define (tail-recursion L acc)
  (if (null? L)
    acc ; Ar trebui aplicat reverse
    (if (= (modulo (car L) 2) 0)
      (tail-recursion (cdr L) (cons
          (car L) acc))
      (tail-recursion (cdr L) acc))))
```

Liste în Racket

```
(member 2 '(1 2 3)) ; => '(2 3)
(member 7 '(1 2 3)) ; => #f
(list-ref (list 'a 'b 'c) 1) ; => 'b
```

Imagini în Racket



```
(overlay                ) ; =>

(flip-vertical          ) ; =>

(ellipse 30 40 "solid" "orange") ; =>
(image-height (ellipse 30 40 "solid"
    "orange")) ; => 40
(image-height (circle 30 "solid" "red"))
    ; => 60
```

Exemplul 3

- **Concatenarea a două liste**

```
(define (app A B)
  (app-iter B (reverse A)))
  ; nevoie de functie ajutatoare
  ; rezultatul este construit in ordine
      inversa

(define (app-iter B Result)
  ; la sfarsit rezultatul e complet
  (if (null? B)
    ; inversam rezultatul
    (reverse Result)
    (app-iter (cdr B) (cons (car B)
        Result))))
    ; construim rezultatul pe avans
```

Important

- apelurile recursive nu consumă spaţiu pe stivă – execuţia este optimizată ştiind că rezultatul apelului recursiv este întors direct, fără operaţii suplimentare;
- complexitate spaţială $O(1)$;
- scriere mai complexă, necesită funcţie auxiliară pentru a avea un parametru suplimentar pentru construcţia rezultatului;
- rezultatul este construit în ordine inversă.

Documentaţie

Pentru mai multe informaţii şi exemple, puteţi consulta documentaţia Racket, disponibilă aici.