Racket: Recursivitate

- Responsabil: Bogdan Popa [mailto:popabogdan97@gmail.com]

- Data publicării: 25.02.2019
- Data ultimei modificări: 08.03.2019 (Soluții)

Obiective

Scopul acestui laborator este studierea diverselor **tipuri** de recursivitate, într-o manieră comparativă, și a modalităților de **eficientizare** a proceselor recursive.

Aspectele urmărite sunt:

- diferențele dintre mai multe **tipuri** de recursivitate, din perspectiva resurselor utilizate și, deci, a eficienței:
  - pe stivă
  - pe coadă
  - arborescentă
- modalități de **transformare** între diversele tipuri.

Arhiva de resurse conține un fișier cu **exerciții rezolvate**, la care textul face trimitere.

Recursivitate

Citiți secțiunea ***1.2.1 Linear Recursion and Iteration** [https://web.mit.edu/alexmv/6.037/sicp.pdf#subsection.1.2.1]* a cărții *Structure and Interpretation of Computer Programs [https://web.mit.edu/alexmv/6.037/sicp.pdf]*, ediția a doua, până la *Example: Counting change*. Apoi, parcurgeți **rezumatul** următor, alături de **exercițiile rezolvate**.

În timpul rulării, un program utilizează o zonă de memorie, denumită **stivă**, cu informațiile de care (mai) are nevoie, în diferite momente. La fiecare **apel** de funcție, spațiul ocupat pe stivă **crește**. La fiecare **revenire** dintr-un apel, cu o anumită valoare, spațiul ocupat pe stivă **scade**. Când recursivitatea este foarte adâncă — există multe apeluri de funcție realizate, din care nu s-a revenit încă — spațiul ocupat pe stivă devine mult prea mare. Acest lucru **influențează**:

- **Memoria**, **principala** resursă afectată. În unele situații, trebuie reținute foarte multe informații. De exemplu, pentru o implementare particulară a funcției `factorial`, ar putea fi necesară reținerea tuturor înmulțirilor care trebuie efectuate la revenirea din recursivitate. Observați exercițiile rezolvate 1 și 2. De obicei, stiva are o dimensiune **maximă**, iar, atunci când aceasta este epuizată, programul se termină brusc, cu o **eroare** de tip „stack overflow”.
- **Timpul**, care, în locul **redimensionării** stivei, ar putea fi utilizat pentru alte calcule, programul rezultat fiind, astfel, mai rapid.

Recursivitate pe stivă

O funcție este recursivă **pe stivă** dacă apelul recursiv este parte a unei expresii mai complexe, fiind necesară **reținerea** de informații, pe stivă, pe avansul în recursivitate.

Exemplu

```
(define (factorial n)
      (if (= n 0)
          1
          (* n (factorial (- n 1)))))
```

Recursivitate pe coadă

O funcție 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 [http://en.wikipedia.org/wiki/Tail_call]*, **nefiind** necesară reținerea de informație pe stivă.

Când o funcție este recursivă **pe coadă**, i se poate aplica ***tail call optimization***. Din moment ce valoarea întoarsă de apelul curent este aceeași cu valoarea întoarsă de apelul recursiv, nu mai este necesară reținerea de informație pe stivă, pentru apelul curent, iar zona aferentă de stivă poate fi **înlocuită**, complet, de cea corespunzătoare apelului recursiv. Astfel, nu mai este necesară stocarea stării fiecărei funcții din apelul recursiv, **spațiul** utilizat fiind **O(1)**. Implementarea de Racket pe care o folosim conține această optimizare, ca parte a specificației.

Optimizarea menționată mai sus se poate aplica și în situații **mai relaxate**, precum *Tail recursion modulo cons [http://en.wikipedia.org/wiki/Tail_call#Tail_recursion_modulo_cons]*. Aici este permisă antrenarea valorii întoarse de apelul recursiv în anumite operații de **construcție**, cum este `cons` pentru liste. Pentru mai multe detalii, citiți informațiile de la adresa precedentă.

Pornind de la cele de mai sus, consumul de resurse poate fi **redus** semnificativ, prin **transformarea** recursivității **pe stivă** în recursivitate **pe coadă** (numită și transformare a recursivității în iterație). Metoda de transformare prezentată în laborator constă în utilizarea unui **acumulator**, ca **parametru** al funcției, în care rezultatul final se construiește treptat, pe **avansul** în recursivitate, în loc de revenire.

Exemplu

```
(define (tail-recursion n acc)
  (if (= n 0)
      acc
      (tail-recursion (- n 1) (* n acc))))

(define (factorial n)
       (tail-recursion n 1))
```

Este important să aveți în vedere faptul că **funcțiile** recursive pe coadă **nu sunt definite** de prezența unui **acumulator**, precum nici funcțiile recursive pe stivă nu sunt definite de absența acestui acumulator. **Diferența** dintre cele două tipuri de funcții constă în **necesitatea** funcțiilor recursive **pe stivă** de a se **întoarce** din recursivitate pentru a **prelucra rezultatul**, respectiv **capacitatea** funcțiilor recursive **pe coadă** de a **genera** un rezultat **pe parcursul** apelului recursiv.

Exemplu

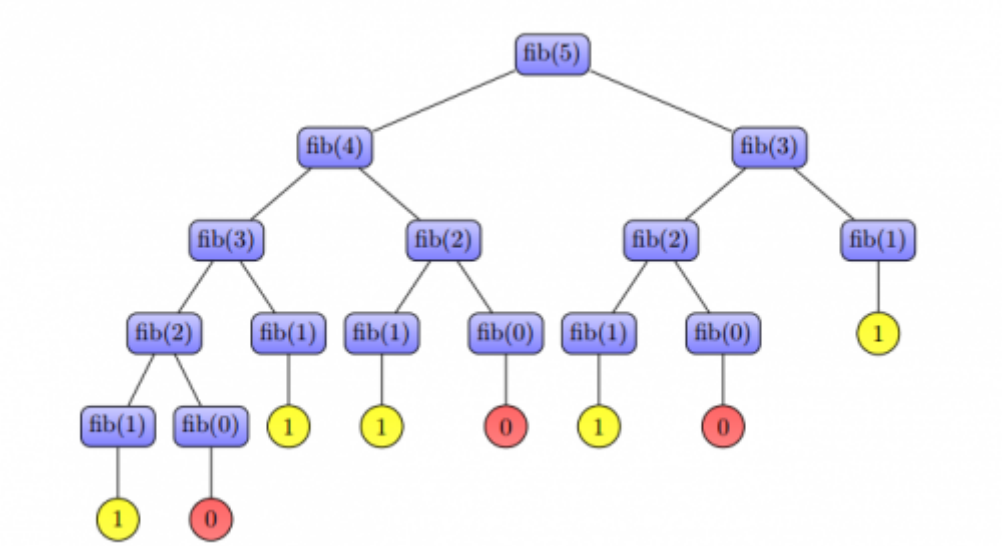
```
(define (member x L)  ;; O posibilă implementare a funcției member
  (if (null? L) #f  ;; Caz de bază, lista vidă nu conține elemente
      (if (equal? x (car L)) (cdr L)  ;; Verificăm dacă elementul căutat este primul termen din listă
          (member x (cdr L))))  ;; Dacă nu, căutăm în restul listei
```

Recursivitate arborescentă

Recursivitatea arborescentă apare în cazul funcțiilor care conțin, în implementare, cel puțin două apeluri recursive care se execută necondiționat.

Exemplu

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```



Se poate observa **ineficiența** implementării recursive pe stivă, atât din punct de vedere **temporal** (datorată **apelurilor duplicate**, precum apelul funcției pentru n = 3), cât și din punct de vedere **spațial** (datorată **stocării de „stack frames“** necesare pentru revenirea din recursivitate).

În cadrul exercițiilor rezolvate puteți observa mai în detaliu diferențele dintre recursivitatea pe stivă si cea pe coada (inclusiv în contextul funcției `fib`).

Resurse

Citiți exercițiile **rezolvate**; apoi, rezolvați exercițiile **propuse**.

- Exerciții rezolvate și propuse
- Cheatsheet (Nu uitați și de cheatsheet-ul de la primul laborator)
- Soluții

Referințe

- *Structure and Interpretation of Computer Programs [https://web.mit.edu/alexmv/6.037/sicp.pdf]*, ediția a doua
- *Tail call [http://en.wikipedia.org/wiki/Tail_call]*
