PARADIGME DE PROGRAMARE

Curs 10

Arhitectura programelor Racket.

Arhitectura programelor Racket – Cuprins

- Motivație
- Optimizări
- Arhitectură
- Macro-uri
- Programare simbolică

Racket: eleganță și control

- Scop: cod elegant, transformat automat în cod eficient
- Provocări (tipice limbajelor funcționale):
  - Alocarea masivă de obiecte mici (=> garbage collection intensiv)
  - Recursivitatea pe stivă (=> consum de memorie)
  - Accesul în timp liniar (O(n) pentru (list-ref L i))
- Optimizări native Racket:
  - Partajare structurală: (cons x L) nu copiază tot, ci conține o referință către L (timp O(1))
  - Modelul contextual: închiderile funcționale „trăiesc” pe heap, nu pe stivă, atât cât e nevoie
  - Garbage collection generațional: colectează foarte rapid obiectele „tinere”, lăsându-le pe cele „bătrâne” (necesare pe termen lung) într-o zonă separată, scanată mai rar
  - Tail-call optimization: transformă recursivitatea pe coadă în iterație (spațiu O(1))

Racket: pentru programatori responsabili

- Sarcina programatorului: cod eficient, care profită de optimizările native
- Regula de aur: Un program Racket eficient:
  - Minimizează alocările
    - combină prelucrări într-o singură parcurgere
  - Protejează stiva
    - recursivitate pe coadă
  - Evită redundanța
    - evaluare întârziată
    - memoizare

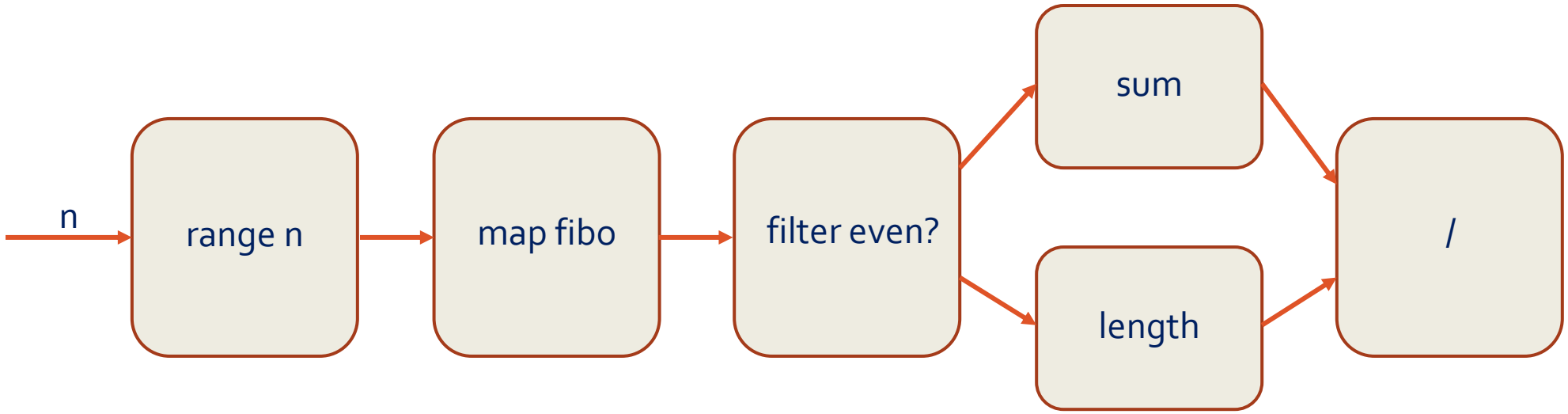
Eficiența programelor Racket – Cuprins

- Motivație
- Optimizări
- Arhitectură
- Macro-uri
- Programare simbolică

Exemplu la calculator

Ex: Să se calculeze media aritmetică a numerelor pare din primele n numere Fibonacci.

O soluție modulară



Analiza soluției

```
1.   (define (fib n)
2.     (if (< n 2)
3.         n
4.         (+ (fib (- n 1)) (fib (- n 2)))))

5.   (define (average-even-fibs n)
6.     (let* ((interval (range n))
7.            (fibs (map fib interval))
8.            (even-fibs (filter even? fibs))
9.            (sum (apply + even-fibs))
10.           (len (length even-fibs)))
11.       (if (zero? len) 0 (/ sum len))))
```

Analiza soluției

```
1.   (define (fib n)
2.     (if (< n 2)
3.         n
4.         (+ (fib (- n 1)) (fib (- n 2)))))

5.   (define (average-even-fibs n)
6.     (let* ((interval (range n))
7.            (fibs (map fib interval))
8.            (even-fibs (filter even? fibs))
9.            (sum (apply + even-fibs))
10.           (len (length even-fibs)))
11.      (if (zero? len) 0 (/ sum len))))
```

→ complexitate exponențială și recursivitate pe stivă

} 3 liste complet construite (multă muncă pentru gc)

} 2 parcurgeri pe aceeași listă

Optimizări automate – insuficiente

```
1.   (define (fib n)
2.     (if (< n 2)
3.         n
4.         (+ (fib (- n 1)) (fib (- n 2)))))

5.   (define (average-even-fibs n)
6.     (let* ((interval (range n))
7.            (fibs (map fib interval))
8.            (even-fibs (filter even? fibs))
9.            (sum (apply + even-fibs))
10.           (len (length even-fibs)))
11.      (if (zero? len) 0 (/ sum len))))
```

gc rapid (identifică listele „de unică folosință”)

partajare structurală (elemente comune fibs/even-fibs)

Probleme rămase

```
1.   (define (fib n)
2.     (if (< n 2)
3.         n
4.         (+ (fib (- n 1)) (fib (- n 2)))))

5.   (define (average-even-fibs n)
6.     (let* ((interval (range n))
7.            (fibs (map fib interval))
8.            (even-fibs (filter even? fibs))
9.            (sum (apply + even-fibs))
10.           (len (length even-fibs)))
11.      (if (zero? len) 0 (/ sum len))))
```

→ complexitate exponențială și recursivitate pe stivă

chiar cu partajare / gc, listele tot se alocă

2 parcurgeri pe aceeași listă

Optimizări propuse

- Complexitate fib
  - ?
- 3 liste alocate (interval, fibs, even-fibs)
  - ?
- 2 parcurgeri even-fibs (sum, len)
  - ?

Optimizări propuse

- Complexitate fib
  - Recursivitate pe coadă
- 3 liste alocate (interval, fibs, even-fibs)
  - Fusion: Un singur fold care fuzionează map și filter
- 2 parcurgeri even-fibs (sum, len)
  - Tupling: Un singur fold care întoarce un tuplu

Observație: și ultimele 2 blocuri pot fuziona.

Analiza soluției optimizate

```
1.  (define (fib n)
2.    (let loop ((n n) (a 0) (b 1))
3.      (if (zero? n) a (loop (- n 1) b (+ a b)))))

4.  (define (average-even-fibs n)
5.    (match-let (((list sum len)
6.                 (foldl (λ (x acc) (let ((f (fib x)))
7.                                     (if (even? f)
8.                                         (list (+ f (car acc)) (+ 1 (cadr acc)))
9.                                         acc)))
10.                       '(0 0)
11.                       (range n))))
12.     (if (zero? len) 0 (/ sum len))))
```

Analiza soluției optimizate

- Complexitate fib
  - ✅ Per apel: timp O(n), spațiu O(1)
  - ❌ În fold, când apelăm fib(x) după fib(x-1), recalculăm tot ⇒ $O(n^2)$ pentru n apeluri
- 3 liste alocate (interval, fibs, even-fibs)
  - ✅ Un singur fold care fuzionează map, filter, și prelucrarea rezultatului
  - ⚠️ Am eliminat o serie de liste intermediare și de parcurgeri,
    dar încă începem cu încărcarea intervalului (range n) în memorie
- 2 parcurgeri even-fibs (sum, len)
  - ✅ Un singur fold care calculează ambele valori este eficient temporal
  - ✅ foldl (recursiv pe coadă) generează un proces eficient spațial
  - ❌ Am pierdut eleganța: modulele sunt amestecate în fold (fib, filtrare, medie)

Fluxuri – eleganță și eficiență?

- Principiu: Modelează o arhitectură de tip „pipeline” folosind fluxuri, câștigând:
  - Modularitate (eleganță, compozabilitate)
    - Codul redevine o succesiune de transformări
    - Ex: fib-stream ⇒ take ⇒ even? ⇒ average
  - Fuziune automată (fusion)
    - Modulele sunt separate conceptual, dar computațional fiecare element parcurge întreg pipeline-ul o singură dată, înainte de a se procesa elementul următor
  - Memoizare
    - Un flux global asigură că elementele deja calculate rămân în memorie
    - Ex: fib-stream(n) profită de memoizarea implicită a fib-stream(n-1) și fib-stream(n-2)
      ⇒ timp O(n) pentru n apeluri, obținut natural, prin structura de date

Analiza soluției optimizate

```
1.   (define fib-stream
2.     (let loop ((a 0) (b 1))
3.       (stream-cons a (loop b (+ a b)))))
4.
5.   (define (average s)
6.     (match (stream-fold (λ (acc x) (list (+ x (car acc)) (+ 1 (cadr acc))))
7.                         '(0 0)
8.                         s)
9.       ((list _ 0) 0)
10.      ((list sum len) (/ sum len))))
11.
12.  (define (average-even-fibs n)
13.    (average
14.     (stream-filter even?
15.                    (stream-take fib-stream n))))
```

Analiza soluției optimizate

- Complexitate `(stream-take fib-stream n)`
  - ✅ O(n)
- Liste alocate
  - ⚠️ Doar cele n valori fibonacci forțate consumă memorie
- Parcurgeri even-fibs (sum, len)
  - ✅ Aceeași soluție eficientă folosind tupling
- Eleganță și expresivitate
  - ✅ Modul average abstractizat ⇒ funcție principală clară, expresivă
  - ⚠️ fib-stream eficient, dar a pierdut corespondența cu formula matematică, recursivă

Eficiența programelor Racket – Cuprins

- Motivație
- Optimizări
- Arhitectură
- Macro-uri
- Programare simbolică

Arhitectură

- Scop: echilibru între expresivitate și eficiență
  - Motto: Utilizatorul trebuie să vadă ce se face, nu cum se optimizează
  - Abordare
    - Decizie: Ce optimizări merită făcute?
    - Abstractizare ierarhică: Ascunde optimizările în tipuri de date sau abstracțiuni procedurale performante
- Optimizări tehnice față de soluția anterioară?
  - memorie O(1) pentru parcurgerea intervalului
    - Dacă fluxul numerelor Fibonacci este ancorat în variabila `fib-stream`, atunci variabila păstrează elementele deja forțate (memorie O(n))
    - Soluție:

```
(stream-take (let loop ((a 0) (b 1))
                  (stream-cons a (loop b (+ a b))))
             n)
```

    - ⚠️ Decizie: Expresivitate sau eficiență maximă?

Arhitectură

- Scop: echilibru între expresivitate și eficiență
  - Motto: Utilizatorul trebuie să vadă ce se face, nu cum se optimizează
  - Abordare
    - Decizie: Ce optimizări merită făcute?
    - Abstractizare ierarhică: Ascunde optimizările în tipuri de date sau abstracțiuni procedurale performante
- Optimizări estetice față de soluția anterioară?
  - Fibonacci recursiv memoizat
    - Eleganța implementării recursive
    - Eficiența implementării cu fluxuri: timp O(n) pentru n apeluri
    - Structura de date care încorporează optimizarea: tabelă de dispersie (hash)
- Decizie: Optimizăm expresivitatea.

Analiza soluției optimizate

```
1.  (define fib-cache (make-hash))
2.  (define (fib n)
3.    (if (< n 2)
4.        n
5.        (hash-ref! fib-cache
6.                   n
7.                   (λ () (+ (fib (- n 1)) (fib (- n 2)))))))

8.  (define (stream-range a b)
9.    (if (= a b)
10.       empty-stream
11.       (stream-cons a (stream-range (add1 a) b))))

12. (define (average-even-fibs n)
13.   (average
14.    (stream-filter even?
15.                   (stream-map fib
16.                               (stream-range 0 n)))))
```

Analiza soluției optimizate

- Eficiență
  - ✅ Timp O(n), spațiu O(n) (spațiu pe heap ocupat de elementele memoizate)
  - ✅ Zero liste lungi (spațiul utilizat de fluxurile neancorate în variabile se eliberează constant)
  - ✅ O singură parcurgere: (fluxuri + tupling)
- Eleganță și expresivitate
  - ✅ Module fib, average, stream-range abstractizate ⇒ funcție principală clară, expresivă
  - ✅ Implementare fib: recursivă, conform specificației matematice
  - ❌ Implementare memoizare
    - vizibilă utilizatorului
    - „murdărește” funcția fib
    - depinde de o variabilă globală (`fib-cache`), ceea ce împiedică analiza și testarea funcției în izolare

Eficiența programelor Racket – Cuprins

- Motivație
- Optimizări
- Arhitectură
- Macro-uri
- Programare simbolică

Macro-uri

Macro

= **funcție care primește cod și returnează cod transformat**

- Motto: dacă limbajul nu are sintaxa dorită, inventeaz-o!
- Mecanism: codul se transformă la compilare, înainte ca programul să ruleze
- Abstractizare:
  - Utilizatorul vede o expresie pură
  - Macro-ul transformă sintaxa pură într-un cod cu detalii impure de implementare
  - Analogie
    - Funcționalele abstractizează șabloane de calcul (procese frecvente)
    - Macro-urile abstractizează șabloane sintactice (structuri de cod repetitive)
      (ex: creează un hash, caută în hash înainte de a face apel recursiv)

Construcția **define-syntax-rule**

```
(define-syntax-rule (nume-macro pattern)
  cod-generat)
```

Componente:	`nume-macro`	= cuvânt cheie pe care îl introducem în limbaj
	`pattern`	= forma codului scris de utilizator
	`cod-generat`	= forma transformată (cod brut rescris, zero evaluări)
Funcționare:	Recunoaștere	Compilatorul vede `nume-macro`
	Potrivire	Leagă variabilele din pattern la expresiile date de utilizator
	Rescriere	Rescrie codul, înlocuind variabilele cu valorile asociate

Macro pentru funcții cu memoizare

```
1.  (define-syntax-rule (define/memo (f args ...) body)
2.    (begin
3.      (define cache (make-hash))
4.      (define (f args ...)
5.        (hash-ref! cache
6.                   (list args ...)
7.                   (λ () body)))))
```

Funcția fib elegantă, cu memoizare

```
1.   (define/memo (fib n)
2.     (if (< n 2)
3.         n
4.         (+ (fib (- n 1)) (fib (- n 2)))))
```

Observații

- Eleganță: aspect identic cu varianta naivă, matematică
- Eficiență: comportament identic cu varianta memoizată (cu sau fără fluxuri)
- Abstractizare: define/memo: concept nou, puternic, reutilizabil
  (ex: probleme de programare dinamică)

Eficiența programelor Racket – Cuprins

- Motivație
- Optimizări
- Arhitectură
- Macro-uri
- Programare simbolică

Homoiconicitate

Homoiconicitate (*homo* = același + *icon* = reprezentare)

= **proprietate a unui limbaj de programare, constând în identitatea dintre structura programului și structura de date internă**

Racket

- Dublă natură a expresiei (f x y)
  - Dată: listă cu 3 simboluri (f, x, y)
  - Cod: apel de funcție (f) pe argumente (x, y)
- Cod Racket = listă care așteaptă să fie procesată
  - Putem parsa și rescrie cod, folosind interfața pentru liste (cons, car, cdr, map, etc.)
  - Rescriere facilă
    - automată – folosind macro-uri
    - manuală – prin funcții definite de utilizator

Prevenirea / lansarea evaluării (quote / eval)

- quote (sau `'`) – îngheață evaluarea expresiei argument
  `'(+ 1 2)` ⇒ `(+ 1 2)`, nu `3`
- quasiquote (sau `` ` ``) – îngheață selectiv evaluarea expresiei argument
- unquote (sau `,`) – evaluează o subexpresie, într-o expresie creată cu quasiquote
  `` `(+ 1 ,(+ 1 1)) `` ⇒ `(+ 1 2)`
- eval – evaluează o expresie „înghețată” într-un context dat
  (efect invers celui de quoting)

```
(eval '(+ 1 2) (make-base-namespace))      ⇒ 3
```

Programare simbolică

Programare simbolică

= **prelucrarea codului ca expresie, fără a se concentra pe valoarea expresiei**

Exemple: prelucrăm expresia `'(+ 1 2)`, nu valoarea `3`
rescriem expresii ca `'(+ 0 x)`, `'(* 1 x)`, înlocuind cu `'x`

Mecanisme de sprijin

- Homoiconicitate ⇒ prelucrare facilă
- Prevenirea evaluării ⇒ permite procesarea ca expresie
- Pattern matching ⇒ izolează codul care trebuie rescris

Exemplu – conversia în named let

```
1.  (define (rewrite-to-named-let expr)
2.    (match expr
3.      ((list 'define (cons f args)
4.             (list 'define (cons g args-g) body)
5.             (cons g-call expressions))
6.       #:when (eq? g g-call) ; g definit = g apelat
7.       `(define ,(cons f args)
8.          (let ,g ,(map list args-g expressions)
9.            ,body)))
10.     (_ expr)))
```

Rezumat

Date eficiente

Procese eficiente

Principii de arhitectură

Facilități pentru rescrierea codului

Macro

Programare simbolică

Funcții pentru prevenirea / lansarea evaluării

Rezumat

Date eficiente: fluxuri (calcul la cerere = economie de resurse, memoizare, fuziune automată)

Procese eficiente

Principii de arhitectură

Facilități pentru rescrierea codului

Macro

Programare simbolică

Funcții pentru prevenirea / lansarea evaluării

Rezumat

Date eficiente: fluxuri (calcul la cerere = economie de resurse, memoizare, fuziune automată)

Procese eficiente: recursivitate pe coadă, funcții cu memoizare

Principii de arhitectură

Facilități pentru rescrierea codului

Macro

Programare simbolică

Funcții pentru prevenirea / lansarea evaluării

Rezumat

Date eficiente: fluxuri (calcul la cerere = economie de resurse, memoizare, fuziune automată)

Procese eficiente: recursivitate pe coadă, funcții cu memoizare

Principii de arhitectură: abstractizare ierarhică: ascunde optimizările sub interfețe expresive, echilibru între expresivitate și performanță

Facilități pentru rescrierea codului

Macro

Programare simbolică

Funcții pentru prevenirea / lansarea evaluării

Rezumat

Date eficiente: fluxuri (calcul la cerere = economie de resurse, memoizare, fuziune automată)

Procese eficiente: recursivitate pe coadă, funcții cu memoizare

Principii de arhitectură: abstractizare ierarhică: ascunde optimizările sub interfețe expresive, echilibru între expresivitate și performanță

Facilități pentru rescrierea codului: macro-uri, programare simbolică

Macro

Programare simbolică

Funcții pentru prevenirea / lansarea evaluării

Rezumat

Date eficiente: fluxuri (calcul la cerere = economie de resurse, memoizare, fuziune automată)

Procese eficiente: recursivitate pe coadă, funcții cu memoizare

Principii de arhitectură: abstractizare ierarhică: ascunde optimizările sub interfețe expresive, echilibru între expresivitate și performanță

Facilități pentru rescrierea codului: macro-uri, programare simbolică

Macro: funcție care primește cod și returnează cod transformat (la compilare)

Programare simbolică

Funcții pentru prevenirea / lansarea evaluării

Rezumat

Date eficiente: fluxuri (calcul la cerere = economie de resurse, memoizare, fuziune automată)

Procese eficiente: recursivitate pe coadă, funcții cu memoizare

Principii de arhitectură: abstractizare ierarhică: ascunde optimizările sub interfețe expresive, echilibru între expresivitate și performanță

Facilități pentru rescrierea codului: macro-uri, programare simbolică

Macro: funcție care primește cod și returnează cod transformat (la compilare)

Programare simbolică: prelucrarea codului ca listă, pentru a-l analiza / optimiza manual

Funcții pentru prevenirea / lansarea evaluării

Rezumat

Date eficiente: fluxuri (calcul la cerere = economie de resurse, memoizare, fuziune automată)

Procese eficiente: recursivitate pe coadă, funcții cu memoizare

Principii de arhitectură: abstractizare ierarhică: ascunde optimizările sub interfețe expresive, echilibru între expresivitate și performanță

Facilități pentru rescrierea codului: macro-uri, programare simbolică

Macro: funcție care primește cod și returnează cod transformat (la compilare)

Programare simbolică: prelucrarea codului ca listă, pentru a-l analiza / optimiza manual

Funcții pentru prevenirea / lansarea evaluării: quote, quasiquote, unquote, eval