

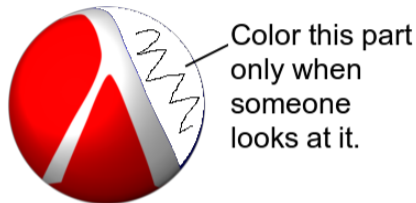
# Paradigme de Programare

Conf. dr. ing. Andrei Olaru

[andrei.olaru@upb.ro](mailto:andrei.olaru@upb.ro) | [cs@andreiolaru.ro](mailto:cs@andreiolaru.ro)  
Departamentul de Calculatoare

2022

## Cursul 5: Evaluare leneșă în Racket



(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101  
103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191  
193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281  
283 293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389  
397 401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 ...



- 1 Întârzierea evaluării
- 2 Fluxuri
- 3 Căutare leneșă în spațiul stărilor

# Întârzierea evaluării



Exemplu

Să se implementeze funcția **nstrictă** *prod*, astfel încât al doilea parametru să fie evaluat doar dacă primul este *true*:

- $prod(F, y) = 0$
- $prod(T, y) = y(y + 1)$

**Dar**, evaluarea parametrului *y* al funcției să se facă numai o singură dată.

· Problema de rezolvat: evaluarea **la cerere**.

# Varianta 1



Încercare → implementare directă

---

```
1 (define prod
2   (lambda (x y)
3     (if x (* y (+ y 1)) 0)))
4
5 (define test
6   (lambda (x)
7     (let ((y 5))
8       (prod x (and (display "y␣") y))))))
9 (test #f)
10 (test #t)
```

Output:

# Varianta 1



Încercare → implementare directă

---

```
1 (define prod
2   (lambda (x y)
3     (if x (* y (+ y 1)) 0)))
4
5 (define test
6   (lambda (x)
7     (let ((y 5))
8       (prod x (and (display "y ") y))))))
9 (test #f)
10 (test #t)
```

Output: y 0 | y 30

- Implementarea nu respectă **specificația**, deoarece **ambii** parametri sunt evaluați în momentul aplicării

# Varianta 2



Încercare → quote & eval

---

```
1 (define prod
2   (lambda (x y)
3     (if x (* (eval y) (+ (eval y) 1)) 0)))
4
5 (define test
6   (lambda (x)
7     (let ((y 5))
8       (prod x (quote (and (display "y ") y))))))
9 (test #f)
10 (test #t)
```

Output:





```
1 (define prod
2   (lambda (x y)
3     (if x (* (eval y) (+ (eval y) 1)) 0)))
4
5 (define test
6   (lambda (x)
7     (let ((y 5))
8       (prod x (quote (and (display "y␣") y))))))
9 (test #f)
10 (test #t)
```

Output: 0 | y undefined

- x = #f → comportament corect: y neevaluat
- x = #t → eroare: quote nu salvează contextul

(

Paranteză!



+ **Context computațional** Contextul computațional al unui punct  $P$ , dintr-un program, la momentul  $t$ , este mulțimea variabilelor ale căror domenii de vizibilitate îl conțin pe  $P$ , la momentul  $t$ .

- Legare **statică** → mulțimea variabilelor care îl conțin pe  $P$  în domeniul **lexical** de vizibilitate
- Legare **dinamică** → mulțimea variabilelor definite cel mai recent, la **momentul**  $t$ , și referite din  $P$



**Ex** Exemplu Ce variabile locale conține contextul computațional al punctului *P*?

```
1 (lambda (x y)
2   (lambda (z)
3     (let ((x (car y)))
4       ; ..P..)))
```



**Ex** Exemplu Ce variabile locale conține contextul computațional al punctului *P*?

```
1 (lambda (x y)
2   (lambda (z)
3     (let ((x (car y)))
4       ; ..P..)))
```



+ **Închidere funcțională:** funcție care își salvează **contextul**, pe care îl va folosi, în momentul **aplicării**, pentru evaluarea corpului.

· **Notatie:** închiderea funcției  $f$  în contextul  $C \rightarrow \langle f; C \rangle$

**Ex** | Exemplu

$\langle \lambda x.z; \{z \leftarrow 2\} \rangle$

)

închidem paranteza

# Varianta 3



## Încercare → închideri funcționale

---

```
1 (define prod
2   (lambda (x y)
3     (if x (* (y) (+ (y) 1)) 0))) ; (y)
4
5 (define test
6   (lambda (x)
7     (let ((y 5))
8       (prod x
9         (lambda () (and (display "y␣") y))))))
10 (test #f)
11 (test #t)
```

Output:



# Varianta 3



## Încercare → închideri funcționale

```
1 (define prod
2   (lambda (x y)
3     (if x (* (y) (+ (y) 1)) 0))) ; (y)
4
5 (define test
6   (lambda (x)
7     (let ((y 5))
8       (prod x
9         (lambda () (and (display "y␣") y))))))
10 (test #f)
11 (test #t)
```

Output: 0 | y y 30

- Comportament corect: y evaluat **la cerere** (deci leneș)
- x = #t → y evaluat de 2 ori → **ineficient**

# Varianta 4



## Promisiuni: delay & force

---

```
1 (define prod
2   (lambda (x y)
3     (if x (* (force y) (+ (force y) 1)) 0)))
4
5 (define test
6   (lambda (x)
7     (let ((y 5))
8       (prod x
9         (delay (and (display "y␣") y))))))
10 (test #f)
11 (test #t)
```

Output:

# Varianta 4



## Promisiuni: delay & force

```
1 (define prod
2   (lambda (x y)
3     (if x (* (force y) (+ (force y) 1)) 0)))
4
5 (define test
6   (lambda (x)
7     (let ((y 5))
8       (prod x
9         (delay (and (display "y ") y))))))
10 (test #f)
11 (test #t)
```

Output: 0 | y 30

- Rezultat corect:  $y$  evaluat **la cerere**, o **singură dată**  
→ **evaluare leneșă** *eficientă*



- Rezultatul încă **neevaluat** al unei expresii
- Valori de **prim rang** în limbaj
- `delay`
  - construiește o promisiune;
  - funcție nestrictă.
- `force`
  - forțează respectarea unei promisiuni, evaluând expresia doar la **prima** aplicare, și **salvându-i** valoarea;
  - începând cu a doua invocare, întoarce, direct, valoarea **memorată**.



- Salvarea **contextului computațional** al expresiei a cărei evaluare este întârziată și evaluarea ei ulterioară în **acel** context → asemănător cu închiderile funcționale.
- Salvarea **rezultatului** primei evaluări a expresiei.
- **Distingerea** primei forțări de celelalte →



- Salvarea **contextului computațional** al expresiei a cărei evaluare este întârziată și evaluarea ei ulterioară în **acel** context → asemănător cu închiderile funcționale.
- Salvarea **rezultatului** primei evaluări a expresiei.
- **Distingerea** primei forțări de celelalte → **efect lateral**, dar acceptabil din moment ce legările se fac static – nu pot exista valori care se schimbă *între timp*.



# Evaluare întârziată

## Abstractizare a implementării cu **promisiuni**

### Ex | Continuare a exemplului cu funcția `prod`

```
1 (define-syntax-rule (pack expr) (delay expr))
2
3 (define unpack force)
4
5 (define prod (lambda (x y)
6   (if x (* (unpack y) (+ (unpack y) 1)) 0)))
7 (define test (lambda (x)
8   (let ((y 5))
9     (prod x (pack (and (display "y ") y)))))))
```

- utilizarea nu depinde de implementare (am definit funcțiile `pack` și `unpack` care **abstractizează** implementarea concretă a evaluării întârziate.



# Evaluare întârziată

## Abstractizare a implementării cu închideri

### Ex) Continuare a exemplului cu funcția prod

```
1 (define-syntax-rule (pack expr) (lambda () expr) )
2
3 (define unpack (lambda (p) (p)))
4
5 (define prod (lambda (x y)
6   (if x (* (unpack y) (+ (unpack y) 1)) 0)))
7 (define test (lambda (x)
8   (let ((y 5))
9     (prod x (pack (and (display "y ") y)))) )))
```

· utilizarea nu depinde de implementare (același cod ca și anterior, altă implementare a funcționalității de evaluare întârziată, acum mai puțin eficientă).



# Fluxuri



**Ex** | Determinați suma numerelor pare<sup>1</sup> din intervalul  $[a, b]$ .

```
1 (define even-sum-iter ; varianta 1
2   (lambda (a b)
3     (let iter ((n a)
4               (sum 0))
5       (cond ((> n b) sum)
6             ((even? n) (iter (+ n 1) (+ sum n)))
7             (else (iter (+ n 1) sum))))))
8
9 (define even-sum-lists ; varianta 2
10  (lambda (a b)
11    (foldl + 0 (filter even? (interval a b)))))
```

---

<sup>1</sup>stă pentru o verificare potențial mai complexă, e.g. numere prime



- Varianta 1 – iterativă (d.p.d.v. proces):
  - **eficientă**, datorită spațiului suplimentar constant;
  - **ne-elegantă** → trebuie să implementăm generarea numerelor.
- Varianta 2 – folosește liste:
  - **ineficientă**, datorită spațiului posibil mare, ocupat la un moment dat – toate numerele din intervalul  $[a, b]$ .
  - **elegantă** și concisă;
- Cum **îmbinăm** avantajele celor 2 abordări? Putem stoca **procesul** fără a stoca **rezultatul** procesului?



- Varianta 1 – iterativă (d.p.d.v. proces):
  - **eficientă**, datorită spațiului suplimentar constant;
  - **ne-elegantă** → trebuie să implementăm generarea numerelor.
- Varianta 2 – folosește liste:
  - **ineficientă**, datorită spațiului posibil mare, ocupat la un moment dat – toate numerele din intervalul  $[a, b]$ .
  - **elegantă** și concisă;
- Cum **îmbinăm** avantajele celor 2 abordări? Putem stoca **procesul** fără a stoca **rezultatul** procesului?



Fluxuri



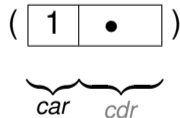
- Secvențe construite **parțial**, extinse la cerere, ce creează **iluzia** completitudinii structurii;
- Îmbinarea **eleganței** manipulării listelor cu **eficiența** calculului incremental;
- Bariera de abstractizare:
  - componentele **listelor** evaluate la **construcție** (`cons`)
  - componentele **fluxurilor** evaluate la **selecție** (`cdr`)
- Construcție și utilizare:
  - **separate** la nivel conceptual → **modularitate**;
  - **întrepătrunse** la nivel de proces (utilizarea necesită construcția concretă).



- o listă este o **pereche**;
- explorarea listei se face prin operatorii `car` – primul element – și `cdr` – **restul** listei;
- am dori să **generăm** `cdr` algoritmic, dar **la cerere**.

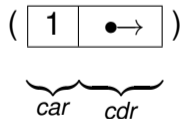


- o listă este o **pereche**;
- explorarea listei se face prin operatorii `car` – primul element – și `cdr` – **restul** listei;
- am dori să **generăm** `cdr` algoritmic, dar **la cerere**.





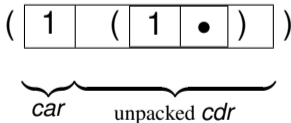
- o listă este o **pereche**;
- explorarea listei se face prin operatorii `car` – primul element – și `cdr` – **restul** listei;
- am dori să **generăm** `cdr` algoritmic, dar **la cerere**.







- o listă este o **pereche**;
- explorarea listei se face prin operatorii `car` – primul element – și `cdr` – **restul** listei;
- am dori să **generăm** `cdr` algoritmic, dar **la cerere**.





- cons, car, cdr, nil, null?

```
1 (define-syntax-rule (stream-cons head tail)
2   (cons head (pack tail)))
3
4 (define stream-car car)
5
6 (define stream-cdr (lambda (s)
7   (unpack (cdr s))))
8
9 (define stream-nil '())
10
11 (define stream-null? null?)
```



- Definiție cu închideri:

```
(define ones (lambda ()(cons 1 (lambda ()(ones))))))
```

- Definiție cu fluxuri:

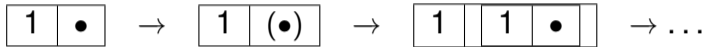
```
1 (define ones (stream-cons 1 ones))  
2 (stream-take 5 ones) ; (1 1 1 1 1)
```

- Definiție cu promisiuni:

```
(define ones (delay (cons 1 ones)))
```



- Ca proces:

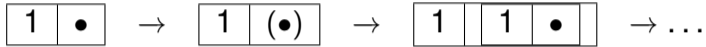


# Fluxuri – Exemple

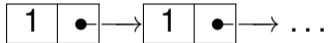
## Flux de numere 1 – discuție



- Ca proces:



- Structural:

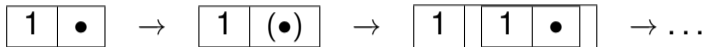


# Fluxuri – Exemple

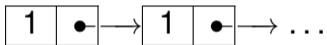
## Flux de numere 1 – discuție



- Ca proces:



- Structural:



- Extinderea se realizează în spațiu constant:

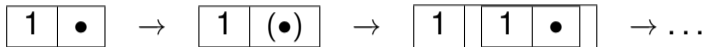


# Fluxuri – Exemple

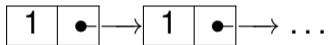
## Flux de numere 1 – discuție



- Ca proces:



- Structural:



- Extinderea se realizează în spațiu constant:

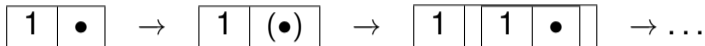


# Fluxuri – Exemple

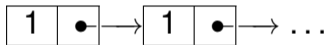
## Flux de numere 1 – discuție



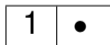
- Ca proces:



- Structural:



- Extinderea se realizează în spațiu constant:



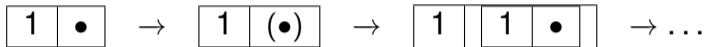


# Fluxuri – Exemple

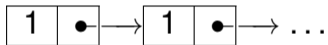
## Flux de numere 1 – discuție



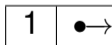
- Ca proces:



- Structural:



- Extinderea se realizează în spațiu constant:

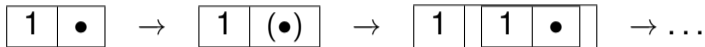


# Fluxuri – Exemple

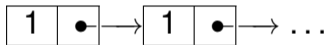
## Flux de numere 1 – discuție



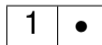
- Ca proces:



- Structural:



- Extinderea se realizează în spațiu constant:





```
1 (define naturals-from (lambda (n)
2   (stream-cons n (naturals-from (+ n 1)))))
3
4 (define naturals (naturals-from 0))

1 (define naturals
2   (stream-cons 0
3     (stream-zip-with + ones naturals)))
```

### · Atenție:

- Închideri: multiple parcurgeri ale fluxului determină **reevaluarea** porțiunilor deja explorate.
- Promisiuni: parcurgerea fluxului determină evaluarea **dincolo** de porțiunile deja explorate.

# Fluxul numerelor pare

În două variante

---



```
1 (define even-naturals
2   (stream-filter even? naturals))
3
4 (define even-naturals
5   (stream-zip-with + naturals naturals))
```



- Ciurul lui **Eratostene**.
- Pornim de la fluxul numerelor **naturale**, începând cu 2.
- Elementul **curent** din fluxul inițial aparține fluxului numerelor prime.
- **Restul** fluxului generat se obține
  - eliminând **multiplii** elementului curent din fluxul inițial;
  - continuând procesul de **filtrare**, cu elementul următor.

# Fluxul numerelor prime

## Implementare

---



```
1 (define sieve (lambda (s)
2   (if (stream-null? s) s
3     (stream-cons (stream-car s)
4       (sieve (stream-filter
5         (lambda (n) (not (zero?
6           (remainder n (stream-car s))))))
7       (stream-cdr s)
8     )))
9 )))
10
11 (define primes (sieve (naturals-from 2)))
```

# Căutare leneșă în spațiul stărilor



+ **Spațiul stărilor unei probleme** Mulțimea configurațiilor valide din universul problemei.



Exemplu

Fie problema  $Pal_n$ : *Să se determine palindroamele de lungime cel puțin  $n$ , ce se pot forma cu elementele unui alfabet fixat.*

**Stările** problemei → **toate** șirurile generabile cu elementele alfabetului respectiv.



# Specificarea unei probleme

Aplicație pe  $Pal_n$



- Starea **inițială**: șirul vid
- Operatorii de generare a stărilor **succesor** ale unei stări: inserarea unui caracter la începutul unui șir dat
- Operatorul de verificare a proprietății de **scop** a unei stări: palindrom



- Spațiul stărilor ca **graf**:
  - noduri: **stări**
  - muchii (orientate): **transformări** ale stărilor în stări succesor
- Posibile strategii de **căutare**:
  - lățime: **completă** și optimală
  - adâncime: **incompletă** și suboptimală



```
1 (define breadth-search-goal
2   (lambda (init expand goal?)
3     (letrec ((search (lambda (states)
4       (if (null? states) '()
5         (let ((state (car states)) (states (cdr states)))
6           (if (goal? state) state
7             (search (append states (expand state))))
8         )))))
9   (search (list init))))
```

- Generarea unei **singure** soluții
- Cum le obținem pe **celelalte**, mai ales dacă spațiul e **infinite**?

# Căutare în lățime

## Leneșă (1) – fluxul stărilor *scop*



```
1 (define lazy-breadth-search (lambda (init expand)
2   (letrec ((search (lambda (states)
3     (if (stream-null? states) states
4       (let ((state (stream-car states))
5           (states (stream-cdr states)))
6         (stream-cons state
7           (search (stream-append states
8                 (expand state))))
9       ))))))
10   (search (stream-cons init stream-nil)))
11 )))
```



```
1 (define lazy-breadth-search-goal
2   (lambda (init expand goal?)
3     (stream-filter goal?
4       (lazy-breadth-search init expand)))
5 ))
```

- Nivel înalt, conceptual: **separare** între explorarea spaţiului şi identificarea stărilor *scop*.
- Nivel scăzut, al instrucţiunilor: **întrepătrunderea** celor două aspecte.
- Aplicaţii:
  - Palindroame
  - Problema regiunilor



- Evaluare întârziată → variante de implementare
- Fluxuri → implementare și utilizări
- Căutare într-un spațiu infinit

+ Dați feedback la acest curs aici:

[[https://docs.google.com/forms/d/e/1FAIpQLSfNqpCfQ25LRn7dL-5BtCeZyVpb9Rd\\_R1u4q0YhWhpReLuWeQ/viewform](https://docs.google.com/forms/d/e/1FAIpQLSfNqpCfQ25LRn7dL-5BtCeZyVpb9Rd_R1u4q0YhWhpReLuWeQ/viewform)]

