

Paradigme de Programare

Conf. dr. ing. Andrei Olaru

andrei.olaru@upb.ro

Departamentul de Calculatoare

2026

Cursul 4: Programare funcțională în Racket



```
(define f (lambda (x) (and (pair? x) (f (cdr x)))) )
```

legare top-level / locală *tipuri* recursivitate

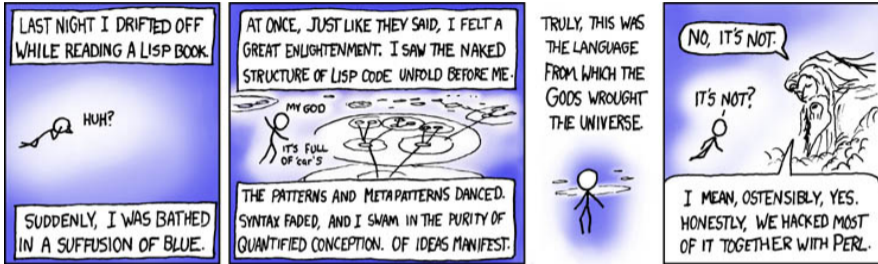


- 1 Introducere
- 2 Legarea variabilelor
- 3 Evaluare
- 4 Construcția programelor prin recursivitate
- 5 Discuție despre tipare

LISP



[(CC) BY-NC xkcd.com] [<https://xkcd.com/224/>]



Introducere



- Gestionarea valorilor
 - modul de tipare al valorilor
 - modul de legare al variabilelor
 - valorile de prim rang

- Gestionarea execuției
 - ordinea de evaluare (generare a valorilor)
 - controlul evaluării
 - modul de construcție al programelor

Legarea variabilelor

⋮ Proprietăți

- identificator
- valoarea legată (la un anumit moment)
- domeniul de vizibilitate (*scope*) + durata de viață
- tip

⋮ Stări

- declarată: cunoaștem **identificatorul**
- definită: cunoaștem și **valoarea** → variabila a fost *legată*

· în Racket, variabilele (numele) sunt legate *static* prin construcțiile `lambda`, `let`, `let*`, `letrec` și `define`, și sunt vizibile în domeniul construcției unde au fost definite (excepție face `define`).

+ **Legarea variabilelor** – modalitatea de **asociere** a apariției unei variabile cu definiția acesteia (deci cu valoarea).

+ **Domeniul de vizibilitate** – *scope* – mulțimea punctelor din program unde o **definiție** (legare) este vizibilă.

+ **Legare statică** – Valoarea pentru un nume este legată o singură dată, **la declarare**, în contextul în care aceasta a fost definită. Valoarea depinde doar de contextul **static** al variabilei.

- Domeniu de vizibilitate al legării poate fi desprins la **compilare**.

+ **Legare dinamică** – Valorile variabilelor depind de **contextul de execuție** în care o expresie este **evaluată**.



- Variabile definite în construcții interioare → **legate static, local**:
 - `lambda`
 - `let`
 - `let*`
 - `letrec`

- Variabile *top-level* → **legate static, global**:
 - `define`



- Leagă **static** parametrii formali ai unei funcții
- Sintaxă:

1 (`lambda` (`p1 ... pk ... pn`) `expr`)

- Domeniul de vizibilitate al parametrului p_k : mulțimea punctelor din `expr` (care este **corpul funcției**), puncte în care apariția lui p_k este **liberă**.



- Aplicație:

```
1 ((lambda (p1 ... pn) expr)
2  a1 ... an)
```

- 1 Evaluare aplicativă: se evaluează **argumentele** a_k , în ordine **aleatoare** (nu se garantează o anumită ordine).
- 2 Se evaluează **corpul** funcției, $expr$, ținând cont de legările $p_k \leftarrow \text{valoare}(a_k)$.
- 3 Valoarea aplicației este **valoarea** lui $expr$, evaluată mai sus.



Construcția `let`

Definiție, Exemplu, Semantică

- Leagă **static** variabile locale
- Sintaxă:

```
1 (let ( (v1 e1) ... (vk ek) ... (vn en) )  
2     expr)
```

- Domeniul de vizibilitate a variabilei v_k (cu valoarea e_k): mulțimea punctelor din `expr` (**corp let**), în care aparițiile lui v_k sunt **libere**.

Ex Exemplu

```
1 (let ((x 1) (y 2)) (+ x 2))
```

- **Atenție!** Construcția `(let ((v1 e1) ... (vn en)) expr)` – **echivalentă** cu `((lambda (v1 ...vn) expr) e1 ...en)`



- Leagă **static** variabile locale
- Sintaxă:

```
1 (let* ((v1 e1) ... (vk ek) ... (vn en))
2   expr)
```

- Scope pentru variabila v_k = mulțimea punctelor din
 - restul **legărilor** (legări ulterioare) și
 - **corp** – expr

în care aparițiile lui v_k sunt **libere**.

Exemplu

```
1 (let* ((x 1) (y x))
2   (+ x 2))
```



```
1 (let* ((v1 e1) ... (vn en))
2   expr)
```

echivalent cu

```
1 (let ((v1 e1))
2   ...
3   (let ((vn en))
4     expr) ... )
```

- Evaluarea expresiilor e_i se face **în ordine!**



- Leagă **static** variabile locale

- Sintaxă:

```
1 (letrec ((v1 e1) ... (vk ek) ... (vn en))
2   expr)
```

- Domeniul de vizibilitate a variabilei v_k = mulțimea punctelor din **întreaga** construcție, în care aparițiile lui v_k sunt **libere**.



Ex Exemplu

```
1 (letrec ((factorial
2         (lambda (n)
3           (if (zero? n) 1
4               (* n (factorial (- n 1)))))))
5   (factorial 5))
```

Construcția define

Definiție & Exemplu



- Leagă **static** variabile **top-level**.
- Avantaje:
 - definirea variabilelor *top-level* în **orice** ordine
 - definirea de funcții **mutual** recursive

Ex Definiții echivalente:

```
1 (define f1
2   (lambda (x)
3     (add1 x)
4   ))
5
6 (define (f2 x)
7   (add1 x)
8 ))
```

Evaluare



- Evaluare **aplicativă**: evaluarea parametrilor **înaintea** aplicării funcției asupra acestora (în ordine aleatoare).
- Funcții **stricte** (i.e. cu evaluare aplicativă)
 - Excepții: `if`, `cond`, `and`, `or`, `quote`.



- quote SAU '
 - funcție **nestrictă**
 - întoarce parametrul **neevaluat**
- eval
 - funcție **strictă**
 - forțează **evaluarea** parametrului și întoarce valoarea acestuia

Ex Exemplu

```
1 (define sum '(+ 2 3))
2 sum ; '(+ 2 3)
3 (eval (list (car sum) (cadr sum) (caddr sum))) ; 5
```

Construcția programelor prin recursivitate



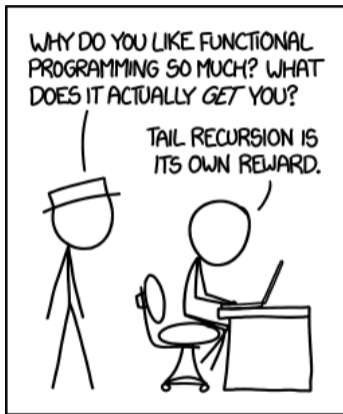
- **Recursivitatea** – element fundamental al paradigmei funcționale
 - Numai prin recursivitate (sau iterare) se pot realiza prelucrări pe date de dimensiuni nedefinite.

- Dar, este eficient să folosim recursivitatea?
 - recursivitatea (pe stivă) poate **încărca stiva**.

Tail recursion



[(CC) BY-NC xkcd.com] [<https://xkcd.com/1270/>]



Alt text: Functional programming combines the flexibility and power of abstract mathematics with the intuitive clarity of abstract mathematics.



- pe stivă: $factorial(n) = n * factorial(n - 1)$
 - timp: liniar
 - spațiu: liniar (ocupat pe stivă)
 - dar, în procedural putem implementa factorialul în spațiu **constant**.



- pe stivă: $factorial(n) = n * factorial(n - 1)$
 - timp: liniar
 - spațiu: liniar (ocupat pe stivă)
 - dar, în procedural putem implementa factorialul în spațiu **constant**.

- pe coadă:
 $factorial(n) = fH(n, 1)$
 $fH(n, p) = fH(n - 1, p * n)$, $n > 1$; p altfel
 - timp: liniar
 - spațiu: constant

- beneficiu *tail call optimization*

Discuție despre tipare



În Racket avem:

- numere: 1, 2, 1.5
- simbolii (literali): 'abcd, 'andrei
- valori booleene: #t, #f
- șiruri de caractere: "șir de caractere"

- perechi: (cons 1 2) → '(1 . 2)
- liste: (cons 1 (cons 2 '())) → '(1 2)

- funcții: (λ (e f) (cons e f)) → #<procedure>

· Cum sunt gestionate tipurile valorilor (variabilelor) la **compilare** (verificare) și la **execuție**?

- Rolul tipurilor: exprimare a intenției programatorului, abstractizare, documentare, optimizare, verificare

+ **Tipare** – modul de gestionare a tipurilor.

· Rolul tipurilor: exprimare a intenției programatorului, abstractizare, documentare, optimizare, verificare

+ **Tipare** – modul de gestionare a tipurilor.

∴ Clasificare după **momentul** verificării:

- statică
- dinamică

∴ Clasificare după **rigiditatea** regulilor:

- tare
- slabă

Ex) Tipare dinamică

Exemplu

Javascript:

```
var x = 5;  
if(condition) x = "here";  
print(x); → ce tip are x aici?
```

Ex) Tipare statică

Exemplu

Java:

```
int x = 5;  
if(condition)  
    x = "here"; → Eroare la compilare: x este int.  
print(x);
```

⋮ Tipare statică

- La compilare
- Valori și variabile
- Rulare mai rapidă

- Rigidă: sancționează orice construcție
- Debugging mai facil
- Declarații explicite sau inferențe de tip
- Pascal, C, C++, Java, Haskell

⋮ Tipare dinamică

- La rulare
- Doar valori
- Rulare mai lentă (necesită verificarea tipurilor)
- Flexibilă: sancționează doar când este necesar
- Debugging mai dificil
- Permite metaprogramare (v. `eval`)
- Python, Scheme/Racket, Prolog, JavaScript, PHP

- Clasificare după libertatea de a agrega valori de tipuri diferite.

Tipare tare

Exemplu $1 + "23" \rightarrow$ Eroare (Haskell, Python)

Tipare slabă

Exemplu $1 + "23" = 24$ (Visual Basic)
 $1 + "23" = "123"$ (JavaScript)



- este **dinamică**

```
1 (if #t 'something (+ 1 #t)) → 'something
```

```
2 (if #f 'something (+ 1 #t)) → Eroare
```

- este **tare**

```
1 (+ "1" 2) → Eroare
```

- dar, permite **liste** cu elemente de tipuri diferite.



- Tipare: dinamică vs. statică, tare vs. slabă;
- Legare: dinamică vs statică;
- Racket: tipare dinamică, tare; domeniu al variabilelor;
- construcții care leagă nume în Racket: `lambda`, `let`, `let*`, `letrec`, `define`;
- evaluare aplicativă;
- construcția funcțiilor prin recursivitate.

+ Dați feedback la acest curs aici:
[\[https://forms.gle/ETsHHPvn2nDgF7q27\]](https://forms.gle/ETsHHPvn2nDgF7q27)

