

# Racket: Legarea variabilelor. Închideri funcționale

- Data publicării: 19.03.2022
- Data ultimei modificări: 19.03.2022

## Obiective

Scopul acestui laborator este prezentarea **conceptelor de legare (binding)** în limbajele din familia LISP, exemplificate în Racket.

Se urmărește stăpânirea conceptelor teoretice:

- **domeniu de vizibilitate**
- **context computațional**
- **legare**
- **legare statică și legare dinamică**
- **închidere funcțională**
- **întârzierea evaluării** (folosind închideri funcționale)

## Domeniu de vizibilitate

în engleză: Scope [[https://en.wikipedia.org/wiki/Scope\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Scope_(computer_science))]

**Domeniul de vizibilitate** al unei variabile este mulțimea punctelor din program în care variabila este vizibilă. Cu alte cuvinte, domeniul de vizibilitate al variabilei *x* este reprezentat de porțiunile din program în care aceasta poate fi accesată (este vizibilă).

**Exemplu:** Domeniul de vizibilitate pentru variabila *a* este format din liniile de cod {9, 10, 11, 12, 13, 14}

```
1 #include<stdio.h>
2
3 int fn() {
4
5 }
6
7
8 int main() {
9     int a = 2, n = 0;
10    for(int i=1; i <= n; i++) {
11        a = a * a;
12    }
13    fn();
14 }
15
```

în afara domeniului de vizibilitate al lui *a* (a nu este vizibil aici)

domeniul de vizibilitate al lui *a* (a este vizibil aici)

## Context computațional

în engleză: Context [[https://en.wikipedia.org/wiki/Context\\_\(computing\)](https://en.wikipedia.org/wiki/Context_(computing))]

Definim **contextul computațional** al unui punct *P* din program la un moment *t* ca fiind mulțimea variabilelor vizibile în punctul *P*, la momentul *t*. Cu alte cuvinte, contextul computațional al unui punct este dat de toate variabilele și valorile acestora, vizibile în acel punct.

**Exemplu:** Pe linia 6 contextul computațional este: {(a 2) (b 32) (s P)}

```
1 #include<stdio.h>
2
3 int main() {
4     int a = 2, b = 32;
5     char s = 'P';
6
7 }
```

Contextul computațional pe linia 6 este:  
(a 2) (b 32) (s 'P')

## Legare

în engleză: Name binding [[https://en.wikipedia.org/wiki/Name\\_binding](https://en.wikipedia.org/wiki/Name_binding)]

O variabilă poate fi reprezentată printr-o pereche dintre un identificator și valoarea acesteia la un moment dat.

```
int random_number = 42;
//
//
//
// "random_number" este valoarea identicatorului.
```

Legarea este procedeul prin care se face asocierea dintre identificatori și locul în care se află valorile efective ale variabilei.

Nu confundați assignment-ul unei variabile cu legarea, citiți această explicație [https://stackoverflow.com/a/48103225].

Legarea poate fi de două tipuri:

- dinamică - toți identificatorii și valorile variabilelor sunt puse într-un context global
- statică - pentru fiecare legare se creează un nou context de identificatori și valori.

În Racket legarea se face prin construcția `let` care permite crearea de variabile ce vor fi vizibile în corpul `let`-ului.

```
(let ((x 2))
  ()); x este vizibil aici si are valoarea 2;))
```

## Legare statică

Este folosită în majoritatea limbajelor de programare din motive istorice (ALGOL 60 / ALGOL 58 / Fortran o folosesc) dar și din motive pragmatice: este ușor de interpretat de către oameni și calculatoare.

Legarea statică creează un nou domeniu de vizibilitate (scope) pentru o variabilă, în funcție de contextul lexical al programului (partea programului care este evaluată), așa că în literatura de specialitate se mai numește **lexical scoping** / **lexical binding**.

În Racket `let` face legare statică:

```
(define (lexical-binding-example x y)
  (let ((x 20)) (+ x y)))
(lexical-binding-example 30 40)
```

2) Construcția `let` va crea un nou context copiindu-l pe cel actual și va adăuga (suprascrie acolo unde este cazul) noile legături:  
`((x 20) (y 40))`

1) Când se apelează funcția se creează un nou context și se adaugă parametri:  
`((x 30) (y 40))`

3) La ieșirea din construcția `let` se va restaura contextul anterior:  
`((x 30) (y 40))`

## Legare dinamică

A fost implementată prima dată în LISP. Este folosită în unele limbaje imperative pentru implementarea variabilelor globale.

În Racket `define` face legare statică globală.

În Scheme (precursorul lui Racket) `define` face legare dinamică:

```
(define a 1)
(define (dynamic-binding-example x)
  (+ x a))
(dynamic-binding-example 2)
(define a 2)
(dynamic-binding-example 2)
```

1) Define-ul va scrie în contextul global:  
`((a 1))`

2) Când se apelează funcția se creează un nou context și se adaugă parametri:  
`((x 2) (a 1))`

3) Define-ul va scrie în contextul global  
`((a 2))`

4) Când se apelează funcția se creează un nou context și se adaugă parametri:  
`((x 2) (a 2))`

Observați că același apel de funcție cu aceiași parametri întoarce rezultate diferite în funcție de contextul global ⇒ **introduce efecte laterale**, de aceea editarea contextului global cu `define` este interzisă în Racket.

## Legare în Racket. Construcții pentru legare

## let

În realitate, un **let** este zăhărel sintactic pentru o  $\lambda$ -expresie. Codul anterior este echivalent cu:

```
((lambda (<id1>...<idn>)
  <expr1>
  ...
  <exprn>)
<val1>
...
<valn>)
```

Corpul unui **let** conține una sau mai multe expresii (<expr1>...<exprn> în exemplul de mai sus). Acestea sunt evaluate în ordine, iar rezultatul întors de **let** este rezultatul evaluării ultimei expresii.

Iată și câteva exemple:

```
(define a 10)

(let ((a 1) (b (+ a 1))) ; aici suntem în zona de definiții, nu în corpul let-ului => a e legat la 10
  (cons a b))           ; în corpul let-ului este vizibilă legarea lui a la 1
```

Codul anterior produce perechea (1 . 11), întrucât legarea (a 1) este vizibilă doar în corpul **let**-ului. a-ul folosit în legarea lui b este 10 mulțumită **define**-ului de mai sus. Fără acel **define**, codul ar fi generat eroare.

```
(let ((a 1)) ; prima legare

  (let ((f (lambda () (print a))))
    (let ((a 2)) ; a doua legare
      (f)))      ; afișează 1
```

În punctul din program corespunzător definirii lui f, identificatorul a este legat la valoarea 1. Faptul că în contextul în care se apelează funcția f a este legat la valoarea 2 nu are importanță. Comparați acest comportament cu exemplul din secțiunea [Legare dinamică](#), de mai sus.

## let\*

Este asemănător cu **let**, însă domeniul de vizibilitate al variabilelor începe imediat după definire. Asta înseamnă că o variabilă definită în **let\*** poate fi folosită în următoarele definiții din cadrul **let\***.

```
(define a 10)

(let* ((a 1) (b (+ a 1))) ; în momentul definirii lui b, este vizibilă legarea lui a la 1
  (cons a b))           ; desigur, aceeași legare e vizibilă și în corpul let-ului
```

Codul anterior întoarce perechea (1 . 2), spre deosebire de (1 . 11) pe care l-am obținut folosind **let**.

## letrec

Domeniul de vizibilitate este întregul **letrec** - cu alte cuvinte inclusiv zona de definire a variabilelor care precede definirea variabilei curente. La momentul în care valoarea variabilei este folosită, ea trebuie să fi fost deja definită (poate fi folosită într-un punct dinaintea definiției sale - din punct de vedere textual - dar nu și din punct de vedere temporal). Acest aspect este ilustrat în exemplele de mai jos:

```
(letrec ((a b) (b 1)) ; în momentul definirii lui a este nevoie de valoarea lui b, necunoscută încă
  (cons a b))        ; de aceea codul produce eroare
```

```
(letrec
  ((even-length?
    (lambda (L)
      (if (null? L) ; even-length? este o funcție, iar
          #t       ; corpul unei funcții nu este evaluat la
          (odd-length? (cdr L)))) ; momentul definirii ei
    (odd-length?
      (lambda (L)
        (if (null? L)
            #f
            (even-length? (cdr L))))))
  (even-length? '(1 2 3 4 5 6))) ; în acest moment deja ambele funcții au fost definite
```

Codul de mai sus întoarce **true**. **odd-length?** este vizibilă în zona de program corespunzătoare definiției lui **even-length?** și va funcționa corect cu condiția ca momentul în care solicităm valoarea sa în acel punct din program să succedă momentului definirii lui **odd-length?**.

## Named let

Această construcție se folosește pentru a obține un mod de a itera în interiorul unei funcții. Numele dat **let**-ului (în cazul de mai jos, **iter**) referă o procedură care primește ca parametri variabilele din lista de definiri și care evaluează expresiile din corpul **let**-ului. Exemplul următor generează intervalul numeric [a, b] cu pasul step (se presupune că diferența dintre b și a este multiplu de step):

```
(define (interval a b step)

  (let iter ((b b) (result '()))          ; parametrul b este inițializat cu valoarea b, result cu '()
    (if (> a b)                             ; aici e vizibil b-ul parametru pentru iter, nu b-ul funcției interval
        result
        (iter (- b step) (cons b result)))) ; iter este apelată recursiv ca o funcție de 2 parametri

(interval 2 10 2) ; întoarce valoarea '(2 4 6 8 10)
```

## Închideri funcționale

Conceptul de închidere funcțională a fost inventat în anii '60 și implementat pentru prima dată în Scheme (din care a fost derivat Racket). Pentru a înțelege acest concept, să ne gândim ce se întâmplă în Racket când definim o funcție, de exemplu funcția de mai sus: `(define f (lambda (x) (+ x a)))`. Ceea ce face orice `define` este să creeze o pereche identificator-valoare; în acest caz se leagă identificatorul `f` la valoarea produsă de evaluarea  $\lambda$ -expresiei `(lambda (x) (+ x a))`. Dar ce valoare produce evaluarea unei  $\lambda$ -expresii? **Evaluarea oricărei  $\lambda$ -expresii produce o închidere funcțională.**

O **închidere funcțională** este o pereche între:

- **textul**  $\lambda$ -expresiei `(lambda (x) (+ x a))` pe exemplul nostru)
- **contextul** computațional în punctul de definire a  $\lambda$ -expresiei `(a 1)` pe exemplul nostru)

Ceea ce salvăm în context este de fapt mulțimea **variabilelor libere** în  $\lambda$ -expresia noastră, adică toate acele variabile referite în textul  $\lambda$ -expresiei dar definite în afara ei. Contextul unei închideri funcționale rămâne cel din momentul creării închiderii funcționale, cu excepția variabilelor definite cu `define`, care ar putea fi înlocuite în timp.

Când o închidere funcțională este aplicată pe argumente, contextul salvat este folosit pentru a da semnificație variabilelor libere din textul  $\lambda$ -expresiei. Este vorba de contextul din momentul aplicării, nu din momentul creării închiderii.

Un aspect remarcabil al închiderilor funcționale este că pot fi folosite pentru a **întârzia evaluarea**. Plecând de la ideea că o închidere funcțională este o pereche text-context, iar textul nu este câtuși de puțin evaluat înainte de aplicarea  $\lambda$ -expresiei pe argumente, consecința este că putem „închide” orice calcul pe care vrem să îl amânăm pe mai târziu într-o expresie `(lambda () calcul)` și să provocăm evaluarea exact la momentul dorit, aplicând această expresie (aici pe 0 argumente).

## Resurse

- Documentație Racket [<https://docs.racket-lang.org/reference/let.html>]
- Schelet [[https://ocw.cs.pub.ro/courses/\\_media/pp/22/laboratoare/racket/legare-schelet.zip](https://ocw.cs.pub.ro/courses/_media/pp/22/laboratoare/racket/legare-schelet.zip)]
- Soluții [[https://ocw.cs.pub.ro/courses/\\_media/pp/22/laboratoare/racket/legare-solutii.zip](https://ocw.cs.pub.ro/courses/_media/pp/22/laboratoare/racket/legare-solutii.zip)]
- Cheatsheet Laborator 4 [<https://github.com/cs-pub-ro/PP-laboratoare/raw/master/racket/legare/legare-cheatsheet.pdf>]

## Referințe

- Lexical Binding [<https://www.cs.oberlin.edu/~bob/cs275.spring14/Examples%20and%20Notes/February/February%2028/Lexical%20and%20Dynamic%20Binding.pdf>]
- Legare statică vs Legare dinamică [<https://www.emacswiki.org/emacs/DynamicBindingVsLexicalBinding>]
- Name, scope, binding [[http://www.cs.iusb.edu/~danav/teach/c311/c311\\_3\\_scope.html](http://www.cs.iusb.edu/~danav/teach/c311/c311_3_scope.html)]