Racket: Legarea variabilelor. Închideri funcționale

- Responsabil: Teodor Szente [mailto:teodor98sz@gmail.com]

- Data publicării: 09.03.2019
- Data ultimei modificări: 09.03.2019

Obiective

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

Se urmărește stăpânirea conceptelor teoretice:

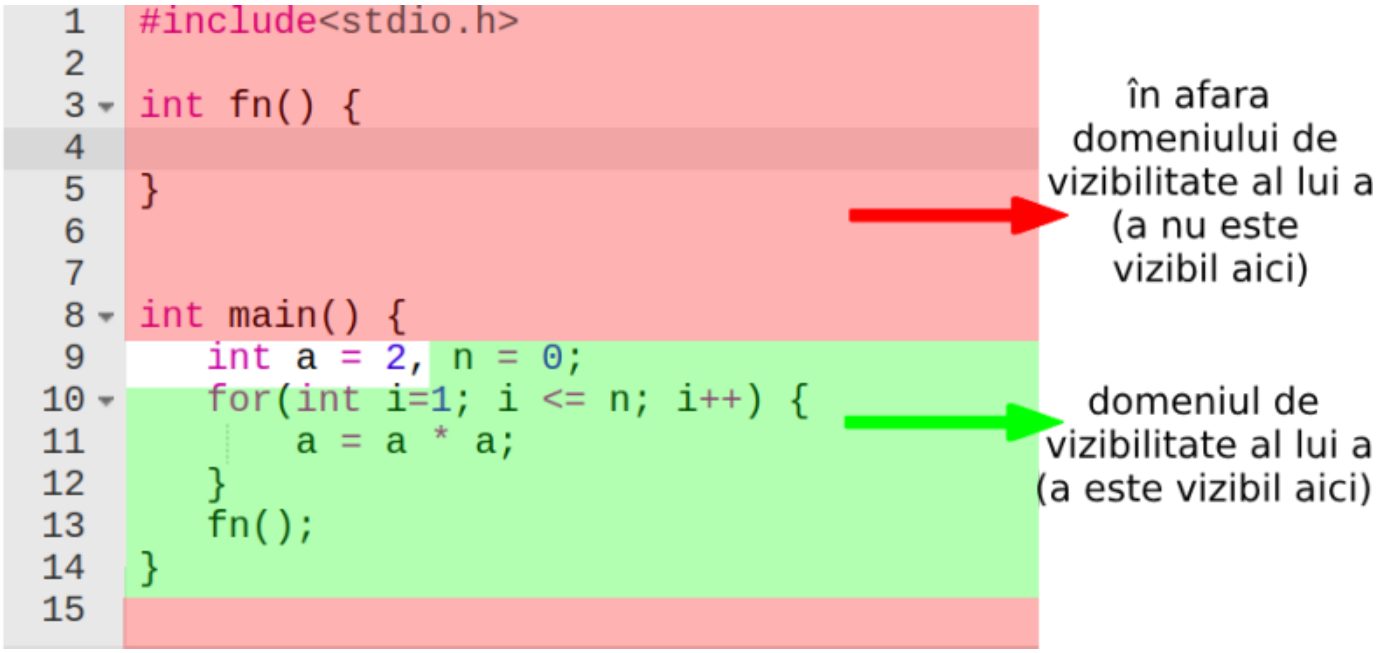
- **domeniu de vizibilitate**
- **context computațional**
- **legare**
- **legare statică** și **legare dinamică**
- **închidere funcțională**
- **întârzierea evaluării** (folosind închideri funcționale)

Domeniu de vizibilitate

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

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}`

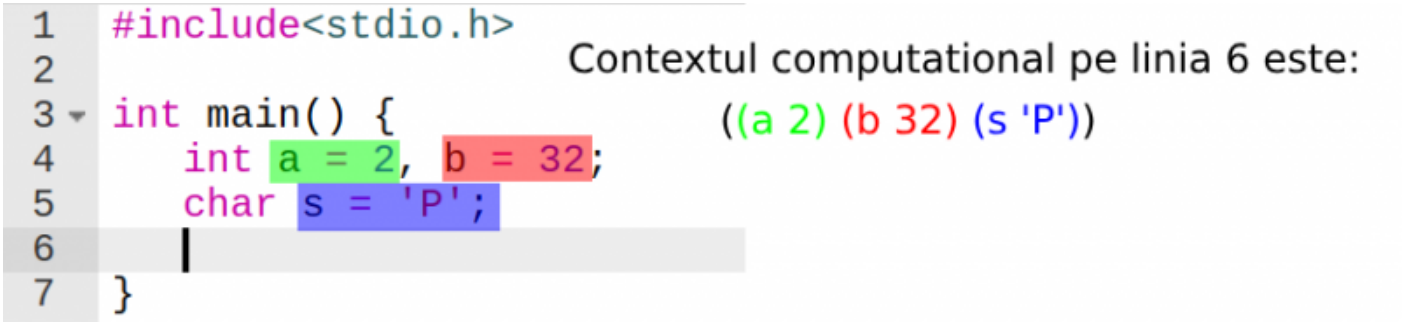


Context computaţional

în engleză: Context [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 şi valorile acestora, vizibile în acel punct.

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



Legare

în engleză: Name binding [https://en.wikipedia.org/wiki/Name_binding]

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

```
int random_number = 42;
//        |          |
//        |          |-------► valoarea variabilei la un moment dat.
//        ▼
//  "random_number" este valoarea identificatorului.
```

Legarea este procedeul prin care se face asocierea dintre identificatori şi 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 şi valorile varabilelor sunt puse într-un context global
- statică - pentru fiecare legare se creează un nou context de identificatori şi valori.

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

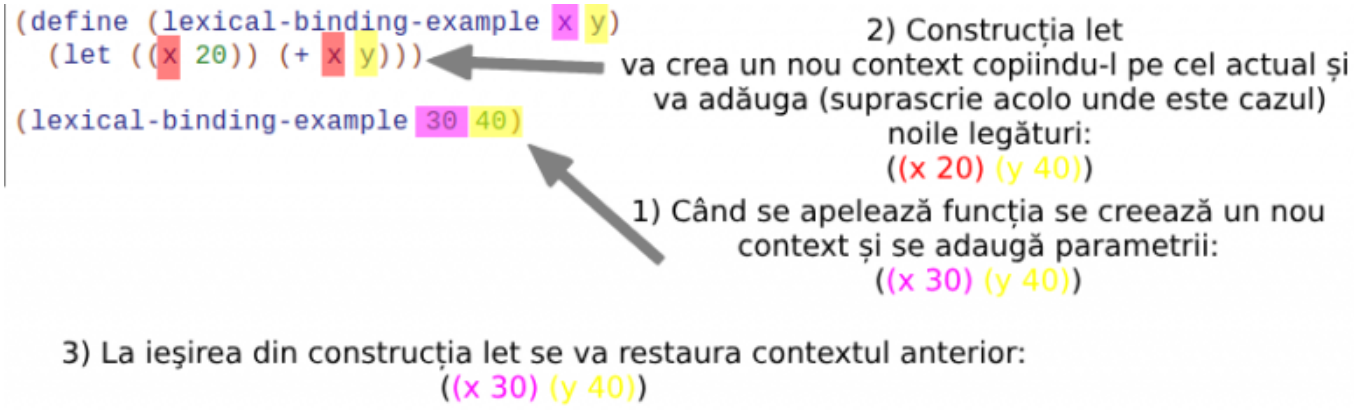
```
(let ((x 2))
    (;; x este vizibil aici si are valoarea 2;;))
```

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**

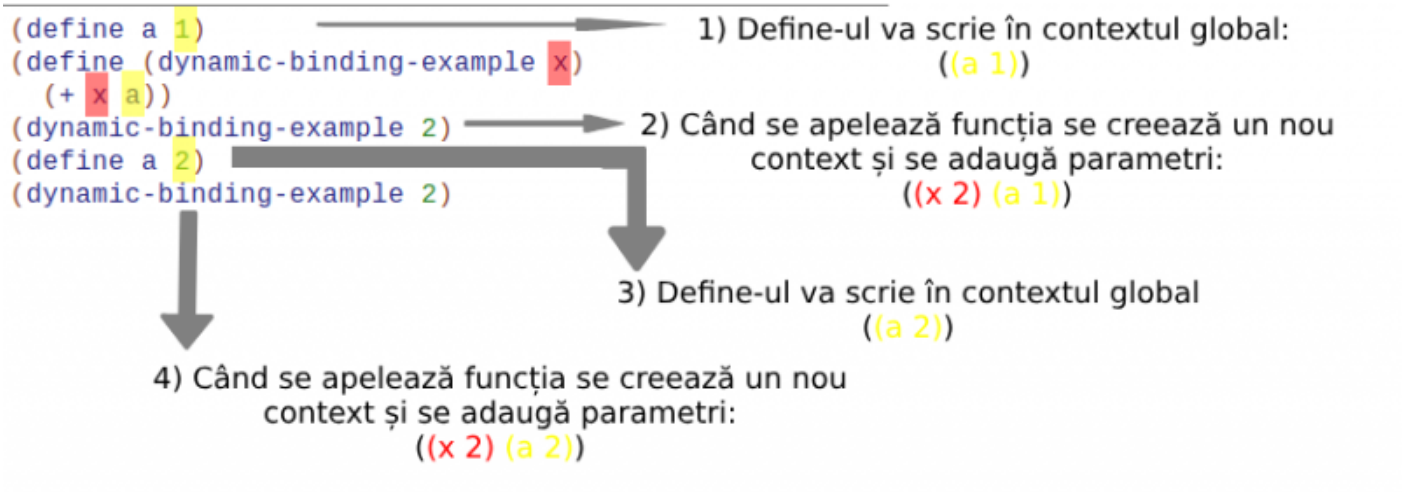
In racket `let` face legare statică:



Legare dinamică

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

În Racket `define` face legare dinamică:



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 in racket. Construcţii pentru legare

let

În realitate, un `let` este zăhărel sintactic pentru o λ-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 in 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 jos.

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 vizbilă 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)                   ; even-length? este o închidere funcțională
        (if (null? L)               ; deci corpul funcției nu este evaluat la
            #t                      ; momentul definirii ei
            (odd-length? (cdr L))))) ; deci nu e o problemă că încă nu știm cine e odd-length?
     (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 succeadă 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 funcţionale

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 λ-expresiei `(lambda (x) (+ x a))`. Dar ce valoare produce evaluarea unei λ-expresii? **Evaluarea oricărei λ-expresii produce o închidere funcțională**.

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

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

Ceea ce salvăm în context este de fapt mulțimea **variabilelor libere** în λ-expresia noastră, adică toate acele variabile referite în textul λ-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 λ-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 căutși de puțin evaluat înainte de aplicarea λ-expresiei pe argumente, consecința este că putem „închide” orice calcul pe care vrem să îl amânăm pe mai târziu într-o expresie (λ () 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]
- Exerciții
- Cheatsheet

Referințe

- Lexical Binding [https://www.cs.oberlin.edu/~bob/cs275.spring14/Examples%20and%20Notes/February/Februrary%2028/Lexical%20and%20Dynamic%20Binding.pdf]
- legare statica vs legare dinamica [https://www.emacswiki.org/emacs/DynamicBindingVsLexicalBinding]
- Name, scope, binding [http://www.cs.iusb.edu/~danav/teach/c311/c311_3_scope.html]