Racket: Introducere

- Responsabil: Mihnea Muraru [mailto:mmihnea@gmail.com]

- Data publicării: 18.02.2019
- Data ultimei modificări: 18.02.2019

Obiective

Scopul acestui laborator este introducerea în **programarea funcțională** și prezentarea elementelor de bază ale limbajului **Racket**.

Aspectele urmărite sunt:

- particularitățile paradigmei **funcționale** în raport cu celelalte paradigme de programare studiate: calculul modelat ca o compunere de funcții, fără secvențe de instrucțiuni, fără cicluri, fără atribuiri (și fără efecte laterale în general)
- programarea în **Racket**: primitivele limbajului, definirea de noi expresii și funcții
- **modelul de evaluare** a expresiilor în Racket: modelul substituției + evaluare aplicativă
- **legătura** TDA (tipuri de date abstracte) - recursivitate - programare funcțională - demonstrații de corectitudine

Particularități ale paradigmelor de programare studiate

Există moduri diferite de a programa un calculator pentru a rezolva o anumită problemă. Vom înțelege prin paradigmă o școală de gândire referitor la organizarea procesului de calcul într-un limbaj de programare.

Vom relua, în ordinea în care au fost studiate, cele două paradigme de programare întâlnite până acum:

- paradigma **procedurală**
- paradigma **orientată obiect**

și vom trece în revistă caracteristicile acestora urmând să prezentăm apoi paradigma **funcțională**.

Programarea procedurală

În programarea procedurală, elementul de bază este **procedura**. Programul constă într-o succesiune de apeluri de proceduri (fie că sunt primitive ale limbajului, fie că sunt definite de programator, caz în care pot conține la rândul lor alte apeluri de proceduri, ș.a.m.d.).

Observăm:

- caracterul **imperativ** al apelurilor: *inițializează(!), calculează(!), dealocă(!)*
- calculele se realizează folosind **efecte laterale** (*side-effects*): Spunem că o procedură are efecte laterale dacă nu doar întoarce un rezultat, ci și modifică starea unor entități din exteriorul ei, de exemplu valorile unor variabile sau structuri reținute în memorie.

Programarea orientată obiect

Programarea orientată obiect mută centrul de interes de la procedură (secvență de prelucrare) la structura de date prelucrată. Elementul de bază este **obiectul**, și fiecare obiect încapsulează metode specifice prin care poate fi modificat/prelucrat.

Și în programarea orientată obiect prelucrarea se bazează pe calcule cu **efecte laterale**: o metodă modifică, de regulă, starea obiectului pe care a fost apelată.

Programarea funcțională

Una dintre principalele diferențe aduse de programarea funcțională este absența **efectelor laterale**, și se datorează faptului că programarea funcțională este atemporală. Nu există atribuiri, nu există secvență de comenzi, o anumită expresie are o singură valoare pe tot parcursul programului. Elementul central este **funcția** (văzută însă nu în sens procedural, ci mai degrabă în sens matematic). Programele constau în compuneri și aplicări de funcții. Exemplu (limbajul `Haskell`):

```
insertion_sort [] = []
insertion_sort (x:xs) = insert x (insertion_sort xs)

insert y [] = [y]
insert y (x:xs) = if y < x then (y:x:xs) else x:(insert y xs)
```

Observați asemănarea izbitoare între codul Haskell și **definirea axiomelor unui TDA**, studiată la cursul de *Analiza Algoritmilor* – funcțiile sunt definite pe cazurile de aplicare și folosesc recursivitatea pentru a referi un caz deja implementat.

Funcția **`insertion_sort`** primește o listă ca parametru, și o sortează prin inserție. Dacă parametrul lui `insertion_sort` este lista vidă, atunci funcția va întoarce lista vidă (care este sortată trivial). Altfel, `insertion_sort` sortează recursiv sublista `xs`, apoi inserează pe poziția corespunzătoare în această listă (sortată) elementul `x`.

Funcția **`insert`** primește doi parametri:

- un element
- o listă sortată.

Dacă lista primită ca parametru este vidă, `insert` întoarce o listă cu un singur element (primul parametru). Altfel, introduce elementul `y` în listă, astfel încât sortarea să se conserve.

Exemplul de mai sus este sugestiv pentru modul de construcție a programelor funcționale: rezultatul final se obține din rezultate intermediare, prin **compuneri și aplicări** de funcții. Definind funcția `insertion_sort` pe o listă nevidă, am observat că avem nevoie întâi să sortăm recursiv lista fără primul element, apoi să inserăm

primul element „la locul lui“ în lista sortată. Programarea funcțională **nu** ne permite secvențe de instrucțiuni de tipul „întâi fă asta, apoi fă cealaltă“, așa că am reformulat secvența de comenzi într-o secvență de aplicări de funcții: „inserează primul element în rezultatul obținut prin sortarea restului“. Nu este nicio problemă că încă nu avem o funcție care inserează un element într-o listă sortată; de fiecare dată când avem nevoie de un rezultat încă necalculat, ne putem **imagina** (*wishful thinking*) că avem deja o funcție care realizează calculul respectiv și putem apela acea funcție, urmând să o implementăm ulterior. Exact așa am procedat cu funcția `insert`. Această abordare ne face să spunem că programarea funcțională este de tip **wishful thinking [https://en.wikipedia.org/wiki/Wishful_thinking]**.

Observați în exemplul de mai sus și **absența** efectelor laterale. Niciuna dintre funcții nu modifică zone de memorie din afara acesteia.

Racket

Racket este un limbaj:

- derivat din **Lisp [https://en.wikipedia.org/wiki/Lisp_(programming_language)]** – sintaxă pe bază de paranteze.
- tipat dinamic – tipul valorilor nu este verificat la compilare.
- tipat tare (strong) – valorile nu se convertesc automat la tipul cerut de situație.
- cu evaluare aplicativă – argumentele funcțiilor sunt evaluate înainte de aplicarea funcției.
- multiparadigmă (suportă programarea funcțională dar are și constructe imperative, pe care noi nu le vom folosi)
- înrudit cu Scheme [http://www.racket-lang.org/new-name.html]

În centrul limbajului Racket se află **evaluarea expresiilor** constând în **aplicări de funcții**. Fie următoarea aplicare a funcției `max`, în **C**:

```
int result;
result = max (3,4);
```

Comparativ, iată aplicarea funcției `max` în `Racket`:

```
(max 3 4)
```

În **C**, apelul de funcție se realizează direct prin nume; acesta este urmat de paranteze, iar între paranteze sunt enumerați parametrii. În **Racket**, paranteza deschisă indică exclusiv faptul că urmează o **aplicare de funcție**. Între paranteze se află numele funcției, urmat de parametri. Următoarea construcție:

```
(max (+ 2 3) 4)
```

se interpretează astfel:

- evaluează aplicarea funcției `max`, care primește doi parametri
- primul parametru reprezintă o altă aplicare a funcției `+` cu parametrii `2` și `3`
- aplicarea funcției `+` se evaluează la `5`
- funcția `max` determină maximul între două numere, iar aplicarea ei pe `5` și `4` se evaluează la 5.

Racket este un limbaj în care argumentele sunt transmise funcției prin valoare (**call-by-value**), astfel că prima expresie evaluată în exemplul de mai sus este (+ 2 3).

Următoarea construcție:

```
(max (3) 4)
```

este **invalidă**. **Racket** va interpreta `(3)` ca pe o tentativă de a aplica funcția cu numele `3` pe zero parametri. Codul va genera eroare.

Exercițiu: încercați să priviți orice construcție din limbajul Racket ca pe o funcție. De exemplu, `if`:

```
(if (= 2 3) 2 (max 2 3))
```

Putem observa faptul că `if` se comportă ca o funcție cu trei parametri:

- primul parametru este **condiția**. Condiția reprezintă o aplicare de funcție, care întoarce (se evaluează la) `true` sau `false`
- al doilea parametru reprezintă expresia de evaluat când condiția este `true`
- al treilea parametru reprezintă expresia de evaluat când condiția este `false`

Cum 2 este diferit de 3, codul de mai sus va întoarce al treilea parametru (evaluarea expresiei `(max 2 3)`).

Tipuri de date

Următoarele tipuri de date sunt uzuale în **Racket** (cu **bold** tipurile pe care le vom folosi mai mult:

- Tipuri de date **simple**
  - **Boolean** – `#t` sau `#f`
  - **Număr**
  - **Simbol** (literal) – `'a`, `'b`, `'ceva`, `'un-simbol-din-mai-multe-cuvinte`, …
  - Caracter
- Tipuri de date **compuse**
  - **Pereche**
  - **Listă**
  - Șir de caractere (String)
  - Vector

Simboluri

Simbolurile (numite și literali) sunt valori formate din unul sau mai multe caractere, fără spațiu. Diferențierea dintre un nume (care este legat la o valoare, similar unei variabile din limbajele imperative) și un simbol se face atașând în fața valorii simbolului un apostrof: `'simbol`.

Atenție! Apostroful în fața unui simbol (sau, vedem mai jos, a unei expresii în paranteză) este un operator, echivalent cu funcția `quote`, care determină ca simbolul sau expresia care îi urmează să nu fie evaluată. Astfel, `'simbol` se evaluează la un simbol și nu se încearcă evaluarea unei variabile cu numele `simbol` și găsirea unei valori asociate acestui nume.

Perechi

O pereche este un tuplu de două elemente, care pot avea tipuri diferite. Pentru manipularea perechilor, Racket ne pune la dispoziție un constructor (`cons`) și doi selectori (`car` și `cdr`). Utilizarea acestora este demonstrată în exemplele de mai jos:

```
(cons 1 2) ; construiește perechea (1 . 2)

(car (cons 1 2)) ; întoarce primul element din pereche, adică 1
(cdr (cons 1 2)) ; întoarce al doilea element din pereche, adică 2

(cons 3 (cons 1 2)) ; construiește PERECHEA (3 . (1 . 2)) (primul element al perechii este un număr, al doilea este o pereche)

(cdr (cons 3 (cons 1 2))) ; întoarce perechea (1 . 2)
```

Liste

Denumirea „Lisp“ a limbajului părinte al Racket-ului provine de la „List Processing“, și într-adevăr lista este o structură de bază în cele două limbaje. Mulțumită faptului că se pot construi perechi eterogene (între elemente de tipuri diferite, de exemplu între un element și o listă), Racket implementează orice listă nevidă ca pe o pereche între primul element și restul listei. Astfel, tipul *listă* **împrumută** de la tipul *pereche* constructorul `cons` și selectorii `car` și `cdr`, la care se adaugă constructorul `null` pentru **lista vidă**. Exemple:

```
(cons 1 null) ; lista formata din elementul 1. Pentru ușurința citirii va fi afișată ca (1) și nu (1 . null), dar reprezintă același lucru
(cons 1 (cons 2 (cons 3 null))) ; lista (1 2 3)
(cons 'a (cons 'b (cons 'c '()))) ; lista (a b c); Atenție, lista vidă se poate reprezenta ca '() sau null;
```

Funcția `list` construiește o listă nouă care va conține elementele date ca argumente funcției:

```
(list 1 2 3 4)
```

Astfel, putem construi lista (1 2 3 4) fie folosind apostroful – `'(1 2 3 4)` – fie folosind funcția `list` – `(list 1 2 3 4)`, dar apostroful nu poate substitui oricând funcția `list`, după cum se observă în exemplele de mai jos:

```
(list 1 2 (+ 2 3)) ; se evaluează la lista (1 2 5)
'(1 2 (+ 2 3)) ; se evaluează la lista (1 2 (+ 2 3)), pentru că întreaga expresie de după apostrof nu se evaluează.
```

Operatori pe liste

```
(car '(1 2 3 4)) ; întoarce 1, adică primul element din perechea formată din elementul 1 și lista (2 3 4)
(cdr '(1 2 3 4)) ; întoarce (2 3 4), adică al doilea element din perechea formată din elementul 1 și lista (2 3 4)
```

Funcțiile `car` și `cdr` pot fi compuse pentru a obține diverse elemente ale listei. Exemple:

```
(car (cdr '(1 2 3 4 5))) ; întoarce 2
(cdr (car '(1 2 3 4 5))) ; cum (car list) nu întoarce o listă, ci un element, apelul produce eroare; funcția cdr așteaptă liste ca parametru
(cdr (cdr '(1 2 3 4 5))) ; întoarce (3 4 5)
(car (cdr (cdr '(1 2 3 4 5))); întoarce 3
```

Racket permite forme prescurtate pentru compuneri de funcții de tip `car` și `cdr`. Rescriem exemplele de mai sus folosind aceste forme prescurtate:

```
(cadr '(1 2 3 4 5)) ; întoarce 2
(cdar '(1 2 3 4 5)) ; produce eroare
(cddr '(1 2 3 4 5)) ; întoarce (3 4 5)
(caddr '(1 2 3 4 5)) ; întoarce 3
```

Alte funcții utile pentru manipularea listelor:

```
(append '(1 2 3) '(4) '(5 6)) ; întoarce lista (1 2 3 4 5 6), din concatenarea tuturor listelor primite ca argumente
(null? '()) ; întoarce #t (adică true), întrucât lista primită ca argument este vidă
(null? '(1 2) ; întoarce #f (adică false), întrucât lista primită ca argument este nevidă
(length '(1 2 3 4)) ; întoarce 4 (lungimea listei primită ca argument)
(length '(1 (2 3) 4)) ; întoarce 3 (lungimea listei primită ca argument)
(reverse '(1 (2 3) 4)) ; întoarce (4 (2 3) 1), adică lista primită ca argument - cu elementele în ordine inversă
(list? '()) ; întoarce #t, întrucât argumentul este o listă
(list? 2) ; întoarce #f, întrucât argumentul nu este o listă
```

Legarea variabilelor

Un identificator poate fi legat la o valoare folosind (printre altele) construcția `(define identificator valoare)`. Efectul `define`-ului este de a permite referirea unei expresii (adesea complexă) cu ajutorul unui nume concis, nu acela de a atribui o valoare unei variabile. În urma `define`-urilor **nu** se suprascriu valori la anumite locații din memorie. Într-un program Racket nu se poate face `define` de mai multe ori la același simbol).

```
(define x 2) ; x devine identificator pentru 2
(define y (+ x 2)) ; y devine identificator pentru 4, întrucât x este doar un alt nume pentru valoarea 2
(define my_list '(a 2 3)) ; my_list identifică lista (a 2 3)

(car my_list) ; intoarce a
(+ (cadr my_list) y) ; întoarce suma dintre al doilea element din lista my_list și y, deci 2 + 4 = 6
```

Funcţii anonime (lambda)

O funcție anonimă se definește utilizând cuvântul cheie `lambda`. Sintaxa este: `(lambda (arg1 arg2 …) ce_întoarce_funcția)`.

```
(lambda (x) x) ; funcția identitate
; pentru a aplica această funcție procedăm în felul următor:
((lambda (x) x) 2) ; întoarce 2; se respectă sintaxa cunoscută (funcție arg1 arg2 ...)

(lambda (x y) (+ x y)) ; funcție care calculează suma a doi termeni x și y
(lambda (l1 l2) (append l2 l1)) ; funcție care concatenează listele l2 și l1, începând cu l2
```

Este destul de neplăcut să rescriem o funcție anonimă pentru a o utiliza în mai multe locuri. Folosim `define` când dorim să legăm un identificator la o funcție anonimă (în aparență funcția nu mai este anonimă).

```
(define identitate (lambda (x) x))
(identitate 3) ; întoarce 3
```

Limbajul ne permite să condensăm definirea unei funcții cu legarea ei la un nume, scriind ca `(define (nume-funcție arg1 arg2 …) ce_întoarce_funcția)`:

```
(define (identitate x) x)
(identitate 3) ; întoarce 3

(define append2 (lambda (l1 l2) (append l2 l1)))
(append2 '(1 2 3) '(4 5 6)) ; întoarce lista (4 5 6 1 2 3)
;fără 'lambda'
(define (append2 l1 l2) (append l2 l1))
(append2 '(1 2 3) '(4 5 6)) ; întoarce lista (4 5 6 1 2 3)
```

Putem oricând scrie `λ` în loc de `lambda` (folosind `Ctrl+\`).

Funcții utile

Operatori:

- aritmetici: `+`, `-`, `*`, `/`, `modulo`, `quotient`
- relaționali: `<`, `<=`, `>`, `>=`, `=`, `eq?`, `equal?`
- logici: `not`, `and`, `or`

```
(modulo 5 2) ; 1, restul împărțirii lui 5 la 2
(quotient 5 2) ; 2, împărțire întreagă

(< 3 2) ; #f
(>= 3 2) ; #t
(= 1 1) ; #t, verifică egalitatea între numere
(= '(1 2) '(1 2)) ; eroare
(equal? '(1 2) '(1 2)) ; #t, verifică egalitatea între valori
(eq? '(1 2 3) '(1 2 3)) ; #f, asemănător cu "==" din Java, verifică dacă două obiecte referă aceeași zonă de memorie

(define x '(1 2 3))
(eq? x x) ; #t
```

Expresii condiționale:

- `if`
- `cond`

```
; (if testexp thenexp elseexp) ; sau fără bucata de else
(if (< a 0)
    a ; întoarce a dacă a este negativ
    (if (> a 10)
        (* a a) ; întoarce a * a dacă a este mai mare decât 10
        0)) ; întoarce 0 altfel

; (cond (test1 exp1) (...) ... (else exp...) ) ; sau fără bucata de else
(cond
   ((< a 0) a) ; întoarce a dacă a este negativ
   ((> a 10) (* a a)) ; întoarce a * a dacă a este mai mare decât 10
   (else 0)) ; întoarce 0 altfel
```

Cum trebuie gândit un program funcțional

Deși Racket este un limbaj multi-paradigmă, veți folosi Racket pentru a învăța să programați în spiritul programării funcționale. Sintetizăm mai jos acest spirit și modul în care puteți suplini lipsa „uneltelor“ cu care v-ați obișnuit în celelalte paradigme:

- programarea este de tip **wishful thinking**
- secvența de instrucțiuni devine **compunere de funcții** (secvență de aplicări de funcții)
- lipsesc atribuirile (variabilă = valoare) și instrucțiunile de ciclare precum `for` sau `while`, dar puteți obține același efect folosind **funcții recursive**. În loc de ciclare și atribuiri (adică în loc să ținem starea curentă a problemei în variabile), vom folosi recursivitate și starea curentă a problemei se va pasa ca parametru în funcțiile recursive. Citiți și recitiți acest paragraf în tandem cu exemplele de mai jos.
- scrierea funcțiilor recursive derivă direct din **scrierea axiomelor** TDA-urilor implicate în problemă

Exemplul 1: o funcție care calculează factorialul unui număr natural n. Știm că TDA-ul Natural are doi constructori de bază, `0` și `succ`. Scriem axiomele operatorului factorial:

```
factorial(0) = 1
factorial(succ(n)) = succ(n) * factorial(n)
```

Traducem întocmai aceste axiome în cod Racket:

```
(define (factorial n) ; identificatorul factorial primește un parametru și anume numărul natural n
      (if (= n 0) ; cazul de bază, n=0, corespunzător primei axiome
          1  ; în acest caz, valoarea întoarsă este 1
          (* n (factorial (- n 1))))) ; altfel, rezultatul este n * factorial(n - 1), corespunzător celei de-a doua axiome

; observați că nu reținem în nicio variabilă n-ul la care am ajuns, dar el este trimis ca parametru dintr-un apel recursiv în altul

(factorial 0) ; întoarce 1
(factorial 1) ; întoarce 1
(factorial 2) ; întoarce 2
(factorial 3) ; întoarce 6
(factorial -1) ; intră în buclă infinită
```

Exemplul 2: o funcție care calculează suma elementelor dintr-o listă L. Știm că TDA-ul List are doi constructori de bază, `null` și `cons`. Scriem axiomele operatorului sum-list:

```
sum-list(null) = 0
sum-list(cons(a,L)) = a + sum-list(L)
```

Trecem axiomele în cod Racket:

```
(define (sum-list L); identificatorul sum-list care primește un parametru, lista L
    (if (null? L) ; dacă L este vidă
        0  ; întoarce 0, cazul corespunzător primei axiome
        (+ (car L) (sum-list (cdr L))))) ; altfel întoarce primul element + sum-list(restul listei), corespunzător celei de-a doua axiome

(sum-list '(1 2 3)) ; întoarce 6
(sum-list 1) ; eroare
```

Resurse

- Exerciții
- Soluții
- Cheatsheet

Referinţe

- Documentație Racket [http://docs.racket-lang.org/reference/index.html], în special funcțiile pentru valori booleene [http://docs.racket-lang.org/reference/booleans.html], numere [http://docs.racket-lang.org/reference/numbers.html] și perechi/liste [http://docs.racket-lang.org/reference/pairs.html]
- Programare funcțională [https://en.wikipedia.org/wiki/Functional_programming]
- Recursivitate [https://en.wikipedia.org/wiki/Recursive]
- Lisp [https://en.wikipedia.org/wiki/Lisp_programming_language]
- Racket [https://en.wikipedia.org/wiki/Racket_(programming_language)]
- Tipare dinamică [https://en.wikipedia.org/wiki/Dynamic_typing#Dynamic_typing]
- Tipare strong [https://en.wikipedia.org/wiki/Strong_typing]
- Evaluare aplicativă [https://en.wikipedia.org/wiki/Eager_evaluation]
- Scheme coding style [https://www.gnu.org/software/mit-scheme/documentation/mit-scheme-user/Coding-style.html]

Is Scheme faster than C? [http://www.cs.indiana.edu/~jsobel/c455-c511.updated.txt]

- /