Paradigme de Programare

Conf. dr. ing. Andrei Olaru

andrei.olaru@cs.pub.ro | cs@andreiolaru.ro
Departamentul de Calculatoare

2020

Cursul 2

Programare funcțională în Racket

Cursul 2: Programare funcțională în Racket

Introducere

Racket vs. Scheme

Cum se numește limbajul despre care discutăm?

- Racket este dialect de Lisp/Scheme (așa cum Scheme este dialect de Lisp);
- Racket este derivat din Scheme, oferind instrumente mai puternice;
- Racket (fost PLT Scheme) este interpretat de mediul DrRacket (fost DrScheme);

[http://en.wikipedia.org/wiki/Racket_(programming_language)]

[http://racket-lang.org/new-name.html]

Analiza limbajului Racket

Ce analizăm la un limbaj de programare?

- Gestionarea valorilor
  - modul de tipare al valorilor
  - modul de legare al variabilelor (managementul valorilor)
  - valorile de prim rang
- Gestionarea execuției
  - ordinea de evaluare (generare a valorilor)
  - controlul evaluării
  - modul de construcție al programelor

Legarea variabilelor

Variabile (Nume)

Proprietăți generale ale 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 $\rightarrow$ 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
Definiții (1)

+ **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.

Legarea variabilelor
Definiții (2)

+ **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 momentul în care o expresie este evaluată. Valoarea poate fi (re-)legată la variabilă ulterior declarării variabilei.

- Domeniu de vizibilitate al unei legări – determinat la execuție.

Legarea variabilelor în Racket

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

Construcția `lambda`
Definiție & Exemplu

- Leagă static parametrii formali ai unei funcții
- Sintaxă:

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

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

Construcția `lambda`
Semantică

- Aplicație:

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

1. Evaluare aplicativă: se evaluează argumentele `ak`, în ordine aleatoare (nu se garantează o anumită ordine).
2. Se evaluează corpul funcției, `expr`, ținând cont de legările `pk` $\leftarrow$ *valoare*(`ak`).
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 `vk` (cu valoarea `ek`): mulțimea punctelor din `expr` (corp let), în care aparițiile lui `vk` 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)`

Construcția `let*`
Definiție & Exemplu

- Leagă static variabile locale
- Sintaxă:

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

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

  în care aparițiile lui `vk` sunt libere.

Ex | Exemplu

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

Construcția `let*`
Semantică

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

echivalent cu

```
(let ((v1 e1))
  ...
    (let ((vn en))
      expr) ... )
```

- Evaluarea expresiilor `ei` se face în ordine!

Construcția `letrec`
Definiție

- Leagă static variabile locale
- Sintaxă:

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

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

Construcția `letrec`
Exemplu

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

Evaluarea în Racket

- 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`.

Controlul evaluării

- `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

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.

Recursivitate
Tipuri

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

Recursivitate
Tipuri

- 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*

Sfârșitul cursului 2
Elemente esențiale

- 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.