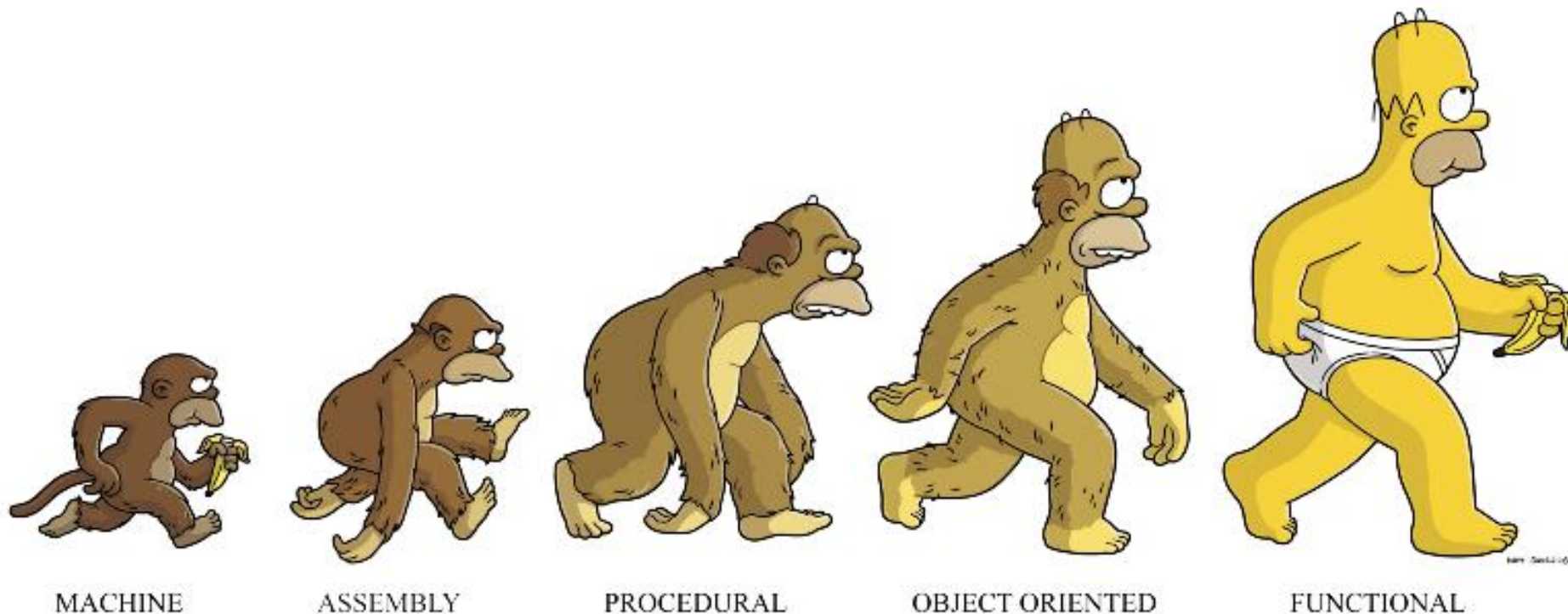


PARADIGME DE PROGRAMARE

Curs 2

Limbajul Racket. Recursivitate.

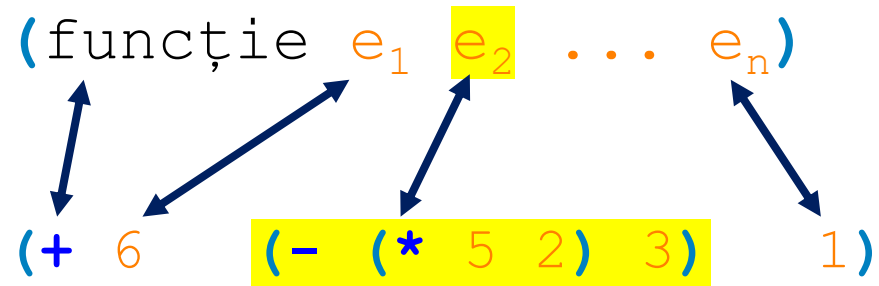
Programare funcțională în Racket



Limbajul Racket – Cuprins

- Expresii și evaluare aplicativă
- Lambda-expresii și funcții
- Perechi și liste
- Valori booleene
- Operatori condiționali
- Recursivitate

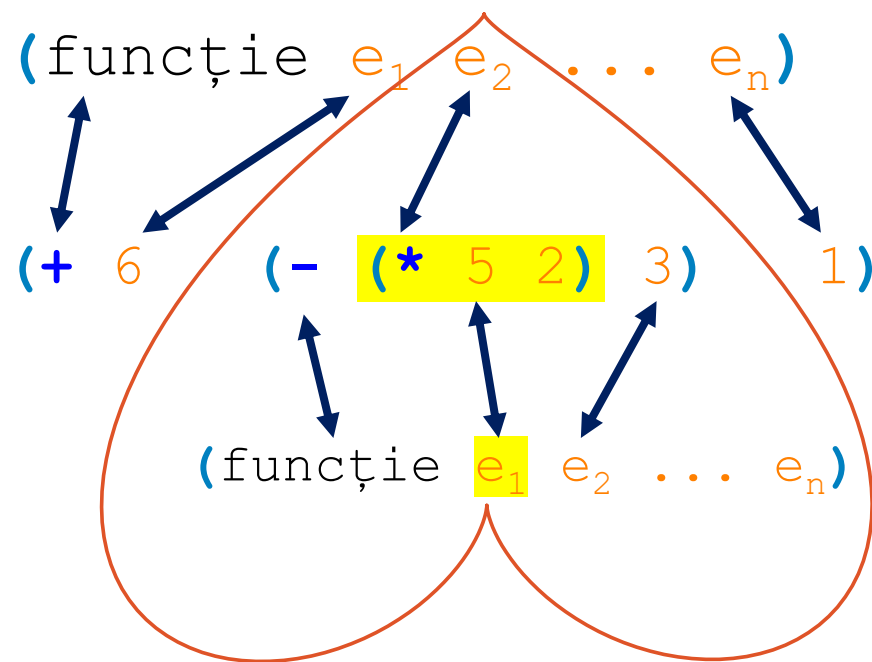
Expresii în Racket



Observație

Fiecare argument al funcției poate fi, la rândul său, o nouă expresie complexă, cu aceeași sintaxă (funcție e_1 e_2 ... e_n). Este cazul lui e_2 de mai sus.

Expresii în Racket



Evaluare aplicativă

Evaluare aplicativă (ex: Racket)

Înainte de a aplica o funcție asupra unor subexpresii, evaluează toate aceste subexpresii cât de mult se poate.



Evaluare normală (ex: Calcul Lambda)

Subexpresiile sunt pasate funcției fără a fi evaluate, în expresia finală subexpresiile reductibile se evaluează de la stânga la dreapta.

Strategii de evaluare

Reprezintă reguli de evaluare a expresiilor într-un limbaj de programare.

Strategii stricte (argumentele funcțiilor sunt evaluate la apel, înainte de aplicare)

- Evaluare aplicativă
- Call by value – funcției i se dă o copie a valorii obținută la evaluare (C, Java, Racket)
- Call by reference – funcției i se pasează o referință la argument (Perl, Visual Basic)

Strategii nestricte (argumentele funcțiilor nu sunt evaluate până ce valoarea lor nu e necesară undeva în corpul funcției)

- Evaluare normală
- Call by name – funcția primește argumentele ca atare, neevaluate, și le evaluează și reevaluează de fiecare dată când valoarea e necesară în corpul funcției
- Call by need – un call by name în care prima evaluare reține rezultatul într-un cache pentru refolosire (Haskell, R)

Construcția **define**

(define identificator **expresie)**

- Creează o pereche identificator-valoare (provenită din evaluarea imediată a **expresiei**), nu alterează o zonă de memorie (≠ atribuire)
- Scopul: **lizibilitate** (numele documentează semnificația valorii)
flexibilitate (la nevoie, valoarea se modifică într-un singur loc)
reutilizare (expresiile complexe nu trebuie rescrise integral de fiecare dată)

Exemple

1. **(define** PI **3.14159265)**

2. **(define** r **5)**

3. **(define** area **(*** PI r r)) ← identificatorul area se leagă la valoarea 78.53981625, fără să rețină de unde a provenit aceasta

Limbajul Racket – Cuprins

- Expresii și evaluare aplicativă
- Lambda-expresii și funcții
- Perechi și liste
- Valori booleene
- Operatori condiționali
- Recursivitate

λ -expresia (în Calculul Lambda)

Sintaxa

$e \equiv$

- | x variabilă
- | $\lambda x.e$ funcție (unară, anonimă) cu parametrul formal x și corpul e
- | $(e_1 e_2)$ aplicație a expresiei e_1 asupra parametrului efectiv e_2

Semantica (Modelul substituției)

Pentru a evalua $(\lambda x.e_1 e_2)$ (funcția cu parametrul formal x și corpul e_1 , aplicată pe e_2):

- Peste tot în e_1 , identificatorul x este înlocuit cu e_2
- Se evaluează noul corp e_1 și se întoarce rezultatul (se notează $e_{1[e_2/x]}$)

Exemple de λ -expresii

$\lambda x.x$

$(x\ y)$

$\lambda x.\lambda y.(x\ y)$

$(\lambda x.x\ a)$

$(\lambda x.y\ a)$

$(\lambda x.x\ \lambda x.y)$

Exemple de λ -expresii

$\lambda x.x$ funcția identitate

$(x\ y)$

$\lambda x.\lambda y.(x\ y)$

$(\lambda x.x\ a)$

$(\lambda x.y\ a)$

$(\lambda x.x\ \lambda x.y)$

Exemple de λ -expresii

$\lambda x.x$

$(x\ y)$ aplicația expresiei x asupra expresiei y

$\lambda x.\lambda y.(x\ y)$

$(\lambda x.x\ a)$

$(\lambda x.y\ a)$

$(\lambda x.x\ \lambda x.y)$

Exemple de λ -expresii

$\lambda x.x$

$(x\ y)$

$\lambda x.\lambda y.(x\ y)$

o funcție de un parametru x care întoarce o altă funcție de un parametru y care îl aplică pe x asupra lui y

$(\lambda x.x\ a)$

$(\lambda x.y\ a)$

$(\lambda x.x\ \lambda x.y)$

Exemple de λ -expresii

$\lambda x.x$

$(x\ y)$

$\lambda x.\lambda y.(x\ y)$

$(\lambda x.x\ a)$ funcția identitate aplicată asupra lui a (se evaluează la a)

$(\lambda x.y\ a)$

$(\lambda x.x\ \lambda x.y)$

Exemple de λ -expresii

$\lambda x.x$

$(x\ y)$

$\lambda x.\lambda y.(x\ y)$

$(\lambda x.x\ a)$

$(\lambda x.y\ a)$ funcția de parametru x cu corpul y , aplicată asupra lui a (se evaluează la y)

$(\lambda x.x\ \lambda x.y)$

Exemple de λ -expresii

$\lambda x.x$

$(x\ y)$

$\lambda x.\lambda y.(x\ y)$

$(\lambda x.x\ a)$

$(\lambda x.y\ a)$

$(\lambda x.x\ \lambda x.y)$ funcția identitate aplicată asupra funcției $\lambda x.y$ (se evaluează la $\lambda x.y$)

Funcții anonime în Racket

`(lambda listă-parametri corp)`

Exemple

$\lambda x.x$ `(lambda (x) x)` ;; echivalent cu $(\lambda (x) x)$

$\lambda x.\lambda y.(x\ y)$ `(lambda (x)`
 `(lambda (y)`
 `(x y)))`

$(\lambda x.x\ \lambda x.y)$ `((lambda (x) x) (lambda (x) y))`

Funcții cu nume în Racket

O funcție este o expresie și, ca orice expresie, poate primi un nume cu **define**.

1. `(define arithmetic-mean`
2. `(λ (x y)`
3. `(/ (+ x y)`
4. `2)))`
5. `(arithmetic-mean 5 19) ;; se evaluează la 12`

Racket permite și sintaxa `(define (nume-funcție x1 x2 ... xn) corp)`:

1. `(define (arithmetic-mean x y) (/ (+ x y) 2))`

Exemplu: evaluare a unei aplicații de funcție

```
1. (define (sum-of-squares x y)
2.   (+ (sqr x) (sqr y)))
3.
4. (sum-of-squares (+ 1 2) (* 3 5))      ;; înlocuiește numele prin valoare
> ((λ (x y) (+ (sqr x) (sqr y))) (+ 1 2) (* 3 5)) ;; evaluare aplicativă
> ((λ (x y) (+ (sqr x) (sqr y))) 3 15)
> ((λ (x y) (+ (sqr x) (sqr y))) 3 15)    ;; modelul substituției (x<-3, y<-15)
> (+ (sqr 3) (sqr 15))                  ;; evaluare aplicativă
> (+ 9 225)
> (+ 9 225)
> 234
```

Evaluare aplicativă versus evaluare normală

- Expresia: `((λ (x y) (+ x x)) (+ 1 2) (/ 1 0))`

- Evaluare aplicativă

```
> ((λ (x y) (+ x x)) (+ 1 2) (/ 1 0)) ;; evaluează argumentele
```

```
> Eroare /: division by zero
```

- Evaluare normală

```
> ((λ (x y) (+ x x)) (+ 1 2) (/ 1 0)) ;; substituie argumentele neevaluate în corp
```

```
> (+ (+ 1 2) (+ 1 2)) ;; evaluează stânga-dreapta
```

```
> (+ 3 (+ 1 2))
```

```
> (+ 3 3)
```

6

Limbajul Racket – Cuprins

- Expresii și evaluare aplicativă
- Lambda-expresii și funcții
- Perechi și liste
- Valori booleene
- Operatori condiționali
- Recursivitate

TDA-ul Pereche

Constructori de bază

cons : $T_1 \times T_2 \rightarrow \text{Pereche}$

// creează o pereche cu punct între orice 2 argumente

Operatori

car : $\text{Pereche} \rightarrow T_1$

// extrage prima valoare din pereche

cdr : $\text{Pereche} \rightarrow T_2$

// extrage a doua valoare din pereche

Exemple

```
(cons (cons 1 2) 'a)
```

```
(cons + 3)
```

```
(car (cons (cons 1 2) 5))
```

```
(cdr '(4 . b))
```

TDA-ul Pereche

Constructori de bază

cons : $T_1 \times T_2 \rightarrow \text{Pereche}$

// creează o pereche cu punct între orice 2 argumente

Operatori

car : $\text{Pereche} \rightarrow T_1$

// extrage prima valoare din pereche

cdr : $\text{Pereche} \rightarrow T_2$

// extrage a doua valoare din pereche

Exemple

(cons (cons 1 2) 'a)

; ; ' ((1 . 2) . a)

(cons + 3)

; ; ' (#<procedure:+> . 3)

(car (cons (cons 1 2) 5))

; ; ' (1 . 2)

(cdr ' (4 . b))

; ; ' b

Sintaxa valorilor de tip Pereche

' (1 . 2) echivalent cu (quote (1 . 2))

Explicație

Funcția **quote** își „citează” argumentul, în sensul că previne evaluarea acestuia. Apostroful este doar o notație prescurtată echivalentă cu funcția **quote**.

Acest artificiu este necesar în reprezentarea valorilor de tip Pereche (sau Listă), pentru că Racket, la întâlnirea unei paranteze deschise, consideră că urmează o funcție și apoi argumentele pe care se aplică aceasta.

Racket va interpreta codul ' (1 2) ca pe lista (1 2), în schimb va da eroare dacă încercăm să rulăm codul (1 2):



```
application: not a procedure;  
expected a procedure that can be applied to arguments  
given: 1  
arguments....:
```

TDA-ul Listă

Constructori (de bază și nu numai)

null : $\rightarrow$ Listă

// creează o listă vidă; echivalent cu valoarea '()

cons : $T \times \text{Listă} \rightarrow \text{Listă}$

// creează o listă prin adăugarea unei valori la începutul unei liste

list : $T_1 \times \dots \times T_n \rightarrow \text{Listă}$

// creează o listă din toate argumentele sale

Operatori

car : Listă $\rightarrow T$

cdr : Listă $\rightarrow \text{Listă}$

null? : Listă $\rightarrow \text{Bool}$

length : Listă $\rightarrow \text{Nat}$

append : Listă $\times$ Listă $\rightarrow \text{Listă}$

Exemple

(**car** (**list** 1 'a +))

(**cdr** ' (2 3 4 5))

(**null?** ' ())

(**length** (**list**))

(**append** (**cons** 1 ' (2)) ' (a b))

TDA-ul Listă

Constructori (de bază și nu numai)

null : $\rightarrow$ Listă

// creează o listă vidă; echivalent cu valoarea '()

cons : $T \times \text{Listă} \rightarrow \text{Listă}$

// creează o listă prin adăugarea unei valori la începutul unei liste

list : $T_1 \times \dots \times T_n \rightarrow \text{Listă}$

// creează o listă din toate argumentele sale

Operatori

car : Listă $\rightarrow T$

cdr : Listă $\rightarrow \text{Listă}$

null? : Listă $\rightarrow \text{Bool}$

length : Listă $\rightarrow \text{Nat}$

append : Listă $\times$ Listă $\rightarrow \text{Listă}$

Exemple

(**car** (**list** 1 'a +))

;; 1

(**cdr** ' (2 3 4 5))

;; ' (3 4 5)

(**null?** ' ())

;; #t

(**length** (**list**))

;; 0

(**append** (**cons** 1 ' (2)) ' (a b))

;; ' (1 2 a b)

Sintaxa valorilor de tip Listă

' (1 2 3) echivalent cu ' (1 . (2 . (3 . ())))

Explicație

Listele sunt reprezentate intern ca perechi (cu punct) între primul element și restul listei.

Așadar:

- lista ' (3) este de fapt o pereche între valoarea 3 și lista vidă: ' (3 . ())
- lista ' (2 3) este o pereche între valoarea 2 și lista ' (3): ' (2 . (3 . ()))
- lista ' (1 2 3) este o pereche între valoarea 1 și lista ' (2 3): ' (1 . (2 . (3 . ())))

Observație

Lista este TDA-ul de bază în programarea funcțională.

Orice funcție Racket are structura unei liste și codul Racket poate fi generat, respectiv parsat în Racket folosind constructori și operatori pe liste.

Limbajul Racket – Cuprins

- Expresii și evaluare aplicativă
- Lambda-expresii și funcții
- Perechi și liste
- Valori booleene
- Operatori condiționali
- Recursivitate

TDA-ul Bool

Constructori de bază

#t : -> Bool // True
#f : -> Bool // False

Operatori

not : $T_1 \rightarrow \text{Bool}$
and : $T_1 \times \dots \times T_n \rightarrow \text{Val}$
or : $T_1 \times \dots \times T_n \rightarrow \text{Val}$

Exemple

(not (null? ' (0))), (not 5)
(and (= 3 4)), (and 'a 'b), (and)
(or #t (/ 1 0)), (or 5 #t), (or)

Observații

- Orice valoare nebooleană este interpretată ca True
- **and** este o funcție nestrictă: întoarce #f la primul #f din listă, altfel întoarce ultima valoare
- **or** este o funcție nestrictă: întoarce prima valoare diferită de #f, altfel întoarce #f

TDA-ul Bool

Constructori de bază

#t : -> Bool // True
#f : -> Bool // False

Operatori

not : $T_1 \rightarrow \text{Bool}$
and : $T_1 \times \dots \times T_n \rightarrow \text{Val}$
or : $T_1 \times \dots \times T_n \rightarrow \text{Val}$

Exemple

`(not (null? ' (0))) , (not 5)                      ;; #t, #f`
`(and (= 3 4)) , (and 'a 'b) , (and)                ;; #f, 'b, #t`
`(or #t (/ 1 0)) , (or 5 #t) , (or)                ;; #t, 5, #f`

Observații

- Orice valoare nebooleană este interpretată ca True
- **and** este o funcție nestrictă: întoarce #f la primul #f din listă, altfel întoarce ultima valoare
- **or** este o funcție nestrictă: întoarce prima valoare diferită de #f, altfel întoarce #f

Limbajul Racket – Cuprins

- Expresii și evaluare aplicativă
- Lambda-expresii și funcții
- Perechi și liste
- Valori booleene
- Operatori condiționali
- Recursivitate

Condiționala **if**

(if condiție rezultat-then rezultat-else)

Exemple

1. **(if (null? '(1 2))** ;; se evaluează la #f
2. **(+ 1 2)** ;; NU se evaluează
3. **(- 7 1))** ;; întreg if-ul se evaluează la 6
4. **(if (< 4 10)** ;; se evaluează la #t
5. **'succes** ;; întreg if-ul se evaluează la 'succes
6. **(/ 'logica 0))** ;; NU se evaluează (de aceea nu dă eroare)

Totul este o funcție

if se comportă ca o funcție cu 3 argumente: condiția, rezultatul pe ramura de then, și rezultatul pe ramura de else.

Întrucât funcția trebuie să se evalueze mereu la ceva, niciunul din cele 3 argumente nu poate lipsi! (nu putem avea un if fără else)

Întrucât unul din argumente nu va fi necesar, **if** nu își evaluează argumentele la apel (este o funcție nestrictă). Evaluarea unui **if** se produce astfel:

- Se evaluează condiția (doar primul argument, nu și celelalte două)
- Dacă rezultatul este true, întregul if este înlocuit cu rezultatul pe ramura de then, altfel întregul if este înlocuit cu rezultatul pe ramura de else
- Se evaluează noua expresie

Temă de gândire: Dacă **if** ar fi strict, recursivitatea ar fi imposibilă. De ce?

Condiționala **cond**

```
(cond (condiție1 rezultat1)  
      ...  
      (condițien rezultatn))
```

← în loc de ultima condiție putem folosi
cuvântul cheie **else**, dar nu e obligatoriu

Exemplu

```
1. (define L '(1 2 3))  
2. (cond  
3.   ((null? L) 0) ;; se evaluează doar condiția la #f  
4.   ((null? (cdr L)) (/ 1 0)) ;; se evaluează doar condiția la #f  
5.   (else 'other)) ;; întregul cond se evaluează la 'other
```

Limbajul Racket – Cuprins

- Expresii și evaluare aplicativă
- Lambda-expresii și funcții
- Perechi și liste
- Valori booleene
- Operatori condiționali
- **Recursivitate**

Recursivitate în programarea funcțională

„Unlearn”:

- Atribuiiri
- Instrucțiuni de ciclare (for, while)
- Secvență de operații (o funcție se evaluează la o unică valoare și nu are efecte laterale)

„Learn”:

Compunere de funcții recursive cu starea problemei pasată ca parametru în aceste funcții



De la axiomele TDA-ului la recursivitate

Exemplu: Suma elementelor dintr-o listă

Axiome

// Operatorul sum

$\text{sum}([]) = 0$

$\text{sum}(x:l) = x + \text{sum}(l)$

Program Racket

```
(define (sum L)
```

```
  (if (null? L)
```

```
      0
```

```
      (+ (car L) (sum (cdr L)))))
```

Observații

Axiomele TDA-ului se traduc direct în cod funcțional

- Trebuie precizat comportamentul funcției pe **toți constructorii de bază**
- Orice **în plus e redundant**
 - ex: e redundant și neelegant să precizez și comportamentul pentru liste de fix un element
- Orice **în minus e insuficient** și duce la eșecul aplicării funcției pe anumite valori
 - ex: $\text{factorial}(1) = 1$; $\text{factorial}(\text{succ}(n)) = \text{succ}(n) * \text{factorial}(n) \Rightarrow$ eroare pentru $\text{factorial}(0)$

Abordare generală în scrierea de funcții recursive

- 1) După ce variabile fac recursivitatea? (ce variabile își schimbă valoarea de la un apel la altul?)
- 2) Scribe condiția de oprire pentru fiecare asemenea variabilă (constructori nulari și externi)
- 3) Scribe ce se întâmplă când problema nu este încă elementară (constructori interni, care generează obligatoriu cel puțin un apel recursiv)

Exemplu: Extragerea primelor n elemente dintr-o listă L

```
(define (take L n)
```

1) După ce variabile fac recursivitatea? (ce variabile își schimbă valoarea de la un apel la altul?)

- Dacă aș ști să extrag din (cdr L), m-ar ajuta? (încerc să scad, pe rând, dimensiunea parametrilor)
- Observ că a lua primele 3 elemente din lista '(1 2 3 4) e totuna cu a lua primele 2 elemente din lista '(2 3 4) și a îl adăuga pe 1 în față
- Rezultă că subproblema care mă ajută are (cdr L) și (- n 1) ca parametri, deci recursivitatea se face atât după L cât și după n

2) Scrie condiția de oprire pentru fiecare asemenea variabilă (constructori nulari și externi)

```
(if (or (zero? n) (null? L))  
    '())
```

3) Scrie ce se întâmplă când problema nu este încă elementară (constructori interni, care generează obligatoriu cel puțin un apel recursiv)

```
(cons (car L) (take (cdr L) (- n 1))))
```


Rezumat

Strategii de evaluare

Lambda-expresii

Sintaxa expresiilor Racket

Sintaxa funcțiilor Racket

Perechi

Liste

Booleeni și operatori condiționali

Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

Rezumat

Strategii de evaluare: strictă (ex: aplicativă), nestrictă (ex: normală)

Lambda-expresii

Sintaxa expresiilor Racket

Sintaxa funcțiilor Racket

Perechi

Liste

Booleeni și operatori condiționali

Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

Rezumat

Strategii de evaluare: strictă (ex: aplicativă), nestrictă (ex: normală)

Lambda-expresii: variabilă – x , funcție – $\lambda x.e$, aplicație – $(e_1 e_2)$

Sintaxa expresiilor Racket

Sintaxa funcțiilor Racket

Perechi

Liste

Booleeni și operatori condiționali

Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

Rezumat

Strategii de evaluare: strictă (ex: aplicativă), nestrictă (ex: normală)

Lambda-expresii: variabilă – x , funcție – $\lambda x.e$, aplicație – $(e_1 e_2)$

Sintaxa expresiilor Racket: (funcție $e_1 e_2 \dots e_n$)

Sintaxa funcțiilor Racket

Perechi

Liste

Booleeni și operatori condiționali

Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

Rezumat

Strategii de evaluare: strictă (ex: aplicativă), nestrictă (ex: normală)

Lambda-expresii: variabilă – x , funcție – $\lambda x.e$, aplicație – $(e_1 e_2)$

Sintaxa expresiilor Racket: (funcție $e_1 e_2 \dots e_n$)

Sintaxa funcțiilor Racket: (lambda ($x_1 x_2 \dots x_n$) corp) sau (define ($f x_1 x_2 \dots x_n$) corp)

Perechi

Liste

Booleeni și operatori condiționali

Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

Rezumat

Strategii de evaluare: strictă (ex: aplicativă), nestrictă (ex: normală)

Lambda-expresii: variabilă – x , funcție – $\lambda x.e$, aplicație – $(e_1 e_2)$

Sintaxa expresiilor Racket: (funcție $e_1 e_2 \dots e_n$)

Sintaxa funcțiilor Racket: (lambda ($x_1 x_2 \dots x_n$) corp) sau (define ($f x_1 x_2 \dots x_n$) corp)

Perechi: $(a . b)$, cons, car, cdr

Liste

Booleeni și operatori condiționali

Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

Rezumat

Strategii de evaluare: strictă (ex: aplicativă), nestrictă (ex: normală)

Lambda-expresii: variabilă – x , funcție – $\lambda x.e$, aplicație – $(e_1 e_2)$

Sintaxa expresiilor Racket: (funcție $e_1 e_2 \dots e_n$)

Sintaxa funcțiilor Racket: (lambda ($x_1 x_2 \dots x_n$) corp) sau (define ($f x_1 x_2 \dots x_n$) corp)

Perechi: $(a . b)$, cons, car, cdr

Liste: $(1 2 3)$, $()$, cons, list, car, cdr, null?, length, append

Booleeni și operatori condiționali

Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

Rezumat

Strategii de evaluare: strictă (ex: aplicativă), nestrictă (ex: normală)

Lambda-expresii: variabilă – x , funcție – $\lambda x.e$, aplicație – $(e_1 e_2)$

Sintaxa expresiilor Racket: (funcție $e_1 e_2 \dots e_n$)

Sintaxa funcțiilor Racket: (lambda ($x_1 x_2 \dots x_n$) corp) sau (define ($f x_1 x_2 \dots x_n$) corp)

Perechi: $(a . b)$, cons, car, cdr

Liste: $(1 2 3)$, $()$, cons, list, car, cdr, null?, length, append

Booleeni și operatori condiționali: #t, #f, and, or, not, if, cond

Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

Rezumat

Strategii de evaluare: strictă (ex: aplicativă), nestrictă (ex: normală)

Lambda-expresii: variabilă – x , funcție – $\lambda x.e$, aplicație – $(e_1 e_2)$

Sintaxa expresiilor Racket: (funcție $e_1 e_2 \dots e_n$)

Sintaxa funcțiilor Racket: (lambda ($x_1 x_2 \dots x_n$) corp) sau (define ($f x_1 x_2 \dots x_n$) corp)

Perechi: $(a . b)$, cons, car, cdr

Liste: $(1 2 3)$, $()$, cons, list, car, cdr, null?, length, append

Booleeni și operatori condiționali: #t, #f, and, or, not, if, cond

Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni: compunere de funcții recursive cu starea problemei pasată ca parametru în aceste funcții