PARADIGME DE PROGRAMARE

Curs 1

Introducere. Modele de evaluare. Limbajul Racket. Recursivitate.

1

Administrative

http://elf.cs.pub.ro/pp/
- Cursuri, laboratoare, teme
- Catalog
- Exemple de examene și teste
- Regulament

Structura fiecărui curs
- Recapitularea cursului anterior
- Predare
- Test din cursul anterior (uneori)
- Rezumatul cursului curent

2

Obiective

Alternative la paradigmele imperativă și orientată obiect
- Paradigma funcțională, paradigma logică

Cum sunt proiectate limbajele de programare
- Modele de calculabilitate
- Features: controlul complexității prin lizibilitate și eficiență
- Limbaje multiparadigmă pentru programatori multiparadigmă

Adaptarea rapidă la noi limbaje de programare
- Racket, Haskell, Prolog

Provocări distractive

3

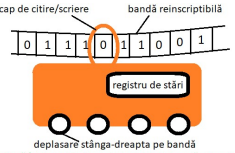
Modele, paradigme, limbaje – Cuprins

- Modele de calculabilitate
- Paradigme de programare
- Exemplu rezolvat în diferite paradigme/limbaje

4

Modele de calculabilitate

Mașina Turing



Dacă în starea 5 citesc simbolul 0 pe bandă, atunci scriu 1, mut capul către stânga și trec în starea 3

Mașina Markov

imperativă -> funcțională
rea -> bună
este cea mai bună -> bate tot

Dându-se s = "Programarea imperativă este cea mai rea" și regulile de substituție de mai sus, asupra lui s se aplică succesiv prima regulă aplicabilă, cât timp ea există.
Programarea funcțională este cea mai rea
Programarea funcțională este cea mai bună
Programarea funcțională bate tot

Calculul Lambda

$(\lambda x.\lambda y.x+y\ 2) \rightarrow \lambda y.2+y$

Dacă aplic funcția cu argumentul **x** și corpul **λy.x+y** asupra valorii **2**, obțin funcția cu argumentul **y** și corpul **2+y**

Mașina Logică

om(alin). om(adina). om(cristi).
place(alin, adina). place(adina, cristi).
fericit(X) dacă om(X), om(Y), place(Y, X).

fericit(Cine).
>>> Cine = adina; Cine = cristi.

Dându-se faptele și regulile anterioare, se încearcă instanțierea variabilelor Cine, X și Y în toate modurile posibile astfel încât să se satisfacă scopul că Cine este fericit.

5

Modele, paradigme, limbaje – Cuprins

- Modele de calculabilitate
- Paradigme de programare
- Exemplu rezolvat în diferite paradigme/limbaje

6

Modele, paradigme, limbaje

Model de calculabilitate
- Oferă un model formal al efectuării calculului
- Diferă de alte modele prin CUM se calculează funcțiile, nu prin CE funcții se calculează

Paradigmă de programare
- Stil fundamental de a programa, bazat pe un anumit model de calculabilitate
- Mod de reprezentare a datelor (ex: variabile, funcții, obiecte, fapte, constrângeri)
- Mod de prelucrare a reprezentării (ex: atribuiri, evaluări, fire de execuție)

Limbaj de programare
- Limbaj formal capabil să exprime procesul de rezolvare a problemelor
- Sprijină una sau mai multe paradigme (ex: Scala, F# - POO și PF; Python – imperativ, POO și PF)

7

Paradigma	Reprezentarea datelor	Structura programului	Execuția programului	Rezultat	Limbaje
Imperativă	Variabile	Succesiune de comenzi	Execuție de comenzi	Stare finală a memoriei	C, Pascal, Fortran
Orientată Obiect	Obiecte	Colecție de clase și obiecte	Transmitere de mesaje între obiecte	Stare finală a obiectelor	Java, C++
Funcțională	Funcții	Colecție de funcții	Evaluare de funcții	Valoarea la care se evaluează funcția principală	Racket, Haskell
Logică	Fapte, reguli	Axiome și o teoremă care trebuie demonstrată	Demonstrarea teoremei	Reușită sau eșec în demonstrarea teoremei	Prolog

8

De ce?

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

Edsger Dijkstra, How do we tell truths that might hurt

I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail.

Abraham Maslow, The law of instrument

The illiterate of the 21st century will not be those who cannot read and write, but those who cannot learn, unlearn, and relearn.

Alvin Toffler

9

Modele, paradigme, limbaje – Cuprins

- Modele de calculabilitate
- Paradigme de programare
- Exemplu rezolvat în diferite paradigme/limbaje

10

Exemplu

Să se determine factorialul unui număr natural n folosind paradigmele:

- Imperativă
- Funcțională
- Logică

11

Rezolvare imperativă

```
1.  int i,  factorial = 1;
2.  for (i = 2; i <= n; i++)
3.      factorial *= i;
```

← datele sunt reținute în **variabilele** i, factorial

← rezolvarea este o **execuție succesivă** de înmulțiri

← rezultatul se regăsește în **starea** finală a memoriei (în zona rezervată variabilei factorial)

De reținut

- Programele imperative au stare
- Starea diferă de la un moment al execuției la altul
- Constructe fundamentale: atribuirea, ciclarea
- Soluție tip „rețetă" (programul descrie CUM se construiește, pas cu pas, rezultatul)

12

Rezolvare funcțională – Racket

```
1.  (define (factorial n)
2.    (if (zero? n)
3.        1
4.        (* n (factorial (- n 1)))))
```

← datele sunt reținute în **funcții** (totul este o funcție)

← caz de bază în **recursivitate** (corespunzător constructorilor nulari/externi)

← apelul recursiv cu o **nouă valoare a parametrului** n (corespunzător constructorilor interni)

De reținut

- Programele funcționale nu au stare
- **Ciclarea** este înlocuită prin **recursivitate**
- **Atribuirea** este înlocuită printr-un apel recursiv cu **noi valori ale parametrilor funcției**
- **Succesiunea de comenzi** este înlocuită prin **compunere de funcții**
- Soluție declarativă (programul descrie CE este, din punct de vedere matematic, rezultatul)

13

Rezolvare funcțională – Haskell

```
1.  factorial 0 = 1
2.  factorial n = n * factorial (n - 1)
```

← totul este o funcție, deci nu este necesar un cuvânt cheie care să spună că urmează o funcție

De observat

- Haskell permite **pattern matching** pe parametrii formali ai funcției (feature existent în Haskell dar nu și în Racket)
- Dacă pattern matching-ul de pe linia 1 reușește, se întoarce rezultatul 1, altfel se încearcă potrivirea cu linia următoare (care, pe codul de mai sus, reușește întotdeauna)

14

Rezolvări funcționale „avansate”

Racket

```
1.  (define (factorial n)
2.    (apply * (range 2 (+ n 1))))
```

Haskell

```
1.  factorial n = product [1 .. n]
```

Pentru rezolvări și mai avansate:

http://www.willamette.edu/~fruehr/haskell/evolution.html

15

Rezolvare logică – Prolog

```
1.  factorial(0, 1).
2.  factorial(N, Result) :-
3.     N > 0,
4.     Prev is N-1,
5.     factorial(Prev, F),
6.     Result is N*F.
```

← datele sunt reținute în **fapte** (ex: factorial de 0 este 1) și **reguli** (ex: factorial de N este Result, dacă:

N > 0 și

factorial de N-1 este F și

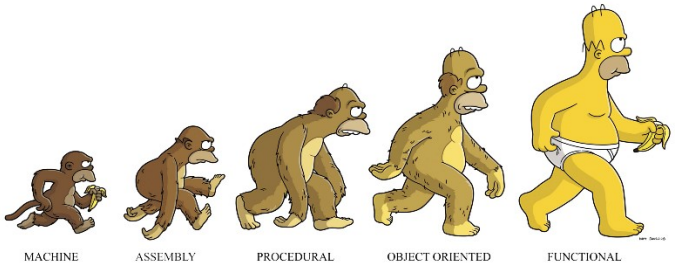
Result este N*F)

De reținut

- Asemănări cu paradigma funcțională: soluție declarativă, ciclarea înlocuită prin recursivitate, atribuirea înlocuită prin noi valori are parametrilor apelului recursiv
- Faptele și regulile sunt axiome, iar o interogare de tip `factorial(5, 120)` sau `factorial(4, F)` reprezintă teorema pe care programul încearcă să o demonstreze
- Limbajul demonstrează teorema potrivind-o în toate modurile posibile cu axiomele existente în universul problemei (**backtracking** încorporat în limbajul de programare)

16

Programare funcțională în Racket



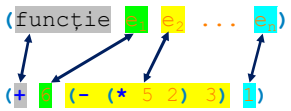
17

Limbajul Racket – Cuprins

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

18

Expresii în Racket

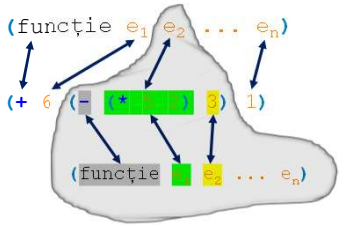


Observație

Fiecare argument al funcției poate fi, la rândul său, o nouă expresie complexă, cu aceeași sintaxă `(funcție e1 e2 ... en)`. Este cazul lui e_2 de mai sus.

19

Expresii în Racket



20

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.

21

Exemplu de evaluare a unei expresii Racket

```
(+ 6 (- (* 5 2) 4))
(+ 6 (- (* 5 2) 4))
(+ 6 (- 10 4))
(+ 6 (- 10 4))
(+ 6 6)
(+ 6 6)
12
```

22

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)

23

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

24

Limbajul Racket – Cuprins

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

25

λ-expresia (în Calculul Lambda)

Sintaxa

e ≡ **x** variabilă

| **λx.e** funcție (unară, anonimă) cu parametrul formal x și corpul e

| **(e1 e2)** aplicație a expresiei e1 asupra parametrului efectiv e2

Semantica (Modelul substituției)

Pentru a evalua **(λx.e1 e2)** (funcția cu parametrul formal x și corpul e1, aplicată pe e2):

- Peste tot în e1, identificatorul x este înlocuit cu e2
- Se evaluează noul corp e1 și se întoarce rezultatul (se notează $e1_{[e2/x]}$)

26

Exemple de λ-expresii

λx.x

(x y)

λx.λy.(x y)

(λx.x a)

(λx.y a)

(λx.x λx.y)

27

Exemple de λ-expresii

λx.x funcția identitate

(x y)

λx.λy.(x y)

(λx.x a)

(λx.y a)

(λx.x λx.y)

28

Exemple de λ-expresii

λx.x

(x y) aplicația expresiei x asupra expresiei y

λx.λy.(x y)

(λx.x a)

(λx.y a)

(λx.x λx.y)

29

Exemple de λ-expresii

λx.x

(x y)

λx.λy.(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

(λx.x a)

(λx.y a)

(λx.x λx.y)

30

Exemple de λ-expresii

λx.x

(x y)

λx.λy.(x y)

(λx.x a) funcția identitate aplicată asupra lui a (se evaluează la a)

(λx.y a)

(λx.x λx.y)

31

Exemple de λ-expresii

λx.x

(x y)

λx.λy.(x y)

(λx.x a)

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

(λx.x λx.y)

32

Exemple de λ-expresii

λx.x

(x y)

λx.λy.(x y)

(λx.x a)

(λx.y a)

(λx.x λx.y) funcția identitate aplicată asupra funcției λx.y (se evaluează la λx.y)

33

Funcții anonime în Racket

```
(lambda listă-parametri corp)
```

Exemple

λx.x
```
(lambda (x) x) ;; echivalent cu (λ (x) x)
```

λx.λy.(x y)
```
(lambda (x)
  (lambda (y)
    (x y)))
```

(λx.x λx.y)
```
((λ (x) x) (λ (x) y))
```

34

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))
```

35

Exemplu de 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
```

36

Limbajul Racket – Cuprins

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

37

TDA-ul Pereche

Constructori de bază

cons : T1 x T2 -> Pereche // creează o pereche cu punct între orice 2 argumente

Operatori

car : Pereche -> T1 // extrage prima valoare din pereche
cdr : Pereche -> T2 // extrage a doua valoare din pereche

Exemple

```
(cons (cons 1 2) 'a)
(cons + 3)
(car (cons (cons 1 2) 5))
(cdr '(4 . b))
```

38

TDA-ul Pereche

Constructori de bază

cons : T1 x T2 -> Pereche // creează o pereche cu punct între orice 2 argumente

Operatori

car : Pereche -> T1 // extrage prima valoare din pereche
cdr : Pereche -> T2 // 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
```

39

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...:
```

40

TDA-ul Listă

Constructori (de bază și nu numai)

null : -> Listă	// creează o listă vidă; echivalent cu valoarea '()
cons : T x Listă -> Listă	// creează o listă prin adăugarea unei valori la începutul unei liste
list : T_1 x .. T_n -> Listă	// creează o listă din toate argumentele sale

Operatori

	Exemple
car : Listă -> T	`(car (list 1 'a +))`
cdr : Listă -> Listă	`(cdr '(2 3 4 5))`
null? : Listă -> Bool	`(null? '())`
length : Listă -> Nat	`(length (list))`
append : Listă x Listă -> Listă	`(append (cons 1 '(2)) '(a b))`

41

TDA-ul Listă

Constructori (de bază și nu numai)

null : -> Listă	// creează o listă vidă; echivalent cu valoarea '()
cons : T x Listă -> Listă	// creează o listă prin adăugarea unei valori la începutul unei liste
list : T_1 x .. T_n -> Listă	// creează o listă din toate argumentele sale

Operatori

	Exemple	
car : Listă -> T	`(car (list 1 'a +))`	`;; 1`
cdr : Listă -> Listă	`(cdr '(2 3 4 5))`	`;; '(3 4 5)`
null? : Listă -> Bool	`(null? '())`	`;; #t`
length : Listă -> Nat	`(length (list))`	`;; 0`
append : Listă x Listă -> Listă	`(append (cons 1 '(2)) '(a b))`	`;; '(1 2 a b)`

42

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.

43

Limbajul Racket – Cuprins

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

44

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)
```

45

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

46

Condiționala `cond`

```
(cond (condiție1 rezultat1)
      ...
      (condiție_n rezultat_n))
```

← 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
```

47

Limbajul Racket – Cuprins

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

48

Recursivitate în programarea funcțională

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

Avem

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

Secvență de operații | Ciclare | Atribuiri

49

De la axiomele TDA-ului la recursivitate

Exemplu: Suma elementelor dintr-o listă

Axiome

// Operatorul sum

sum([ ]) = 0

sum(x:l) = x + sum(l)

Program Racket

```
(define (sum L)
 (if (null? L)
  0
  (+ (car L) (sum (cdr L)))))
```

50

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: factorial(1) = 1; factorial (succ(n)) = succ(n) * factorial(n) => eroare pentru 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) Scrie condiția de oprire pentru fiecare asemenea variabilă (constructori nulari și externi)
3) Scrie ce se întâmplă când problema nu este încă elementară (constructori interni, care generează obligatoriu cel puțin un apel recursiv)

51

Exemplu: Extragerea primelor n elemente dintr-o listă L

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

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 n cât și după L
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 (- n 1) (cdr L)))))
```

52

Rezumat

Modele de calculabilitate
Paradigme
Strategii de evaluare
Lambda-expresii
Sintaxa expresiilor Racket
Sintaxa funcțiilor Racket
Perechi și Liste
Operatori condiționali
Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

53

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda, Mașina Markov, Mașina Logică
Paradigme
Strategii de evaluare
Lambda-expresii
Sintaxa expresiilor Racket
Sintaxa funcțiilor Racket
Perechi și Liste
Operatori condiționali
Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

54

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda, Mașina Markov, Mașina Logică
Paradigme: imperativă, orientată obiect, funcțională, logică
Strategii de evaluare
Lambda-expresii
Sintaxa expresiilor Racket
Sintaxa funcțiilor Racket
Perechi și Liste
Operatori condiționali
Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

55

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda, Mașina Markov, Mașina Logică
Paradigme: imperativă, orientată obiect, funcțională, logică
Strategii de evaluare: strictă (ex: aplicativă), nestrictă (ex: normală)
Lambda-expresii
Sintaxa expresiilor Racket
Sintaxa funcțiilor Racket
Perechi și Liste
Operatori condiționali
Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

56

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda, Mașina Markov, Mașina Logică
Paradigme: imperativă, orientată obiect, funcțională, logică
Strategii de evaluare: strictă (ex: aplicativă), nestrictă (ex: normală)
Lambda-expresii: variabilă (x), funcție (λx.e), aplicație ((e1 e2))
Sintaxa expresiilor Racket
Sintaxa funcțiilor Racket
Perechi și Liste
Operatori condiționali
Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

57

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda, Mașina Markov, Mașina Logică
Paradigme: imperativă, orientată obiect, funcțională, logică
Strategii de evaluare: strictă (ex: aplicativă), nestrictă (ex: normală)
Lambda-expresii: variabilă (x), funcție (λx.e), aplicație ((e_1 e_2))
Sintaxa expresiilor Racket: (funcție e_1 e_2 ... e_n)
Sintaxa funcțiilor Racket:
Perechi și Liste
Operatori condiționali
Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

58

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda, Mașina Markov, Mașina Logică
Paradigme: imperativă, orientată obiect, funcțională, logică
Strategii de evaluare: strictă (ex: aplicativă), nestrictă (ex: normală)
Lambda-expresii: variabilă (x), funcție (λx.e), aplicație ((e_1 e_2))
Sintaxa expresiilor Racket: (funcție e_1 e_2 ... e_n)
Sintaxa funcțiilor Racket: (lambda (x_1 x_2 ... x_n) corp) sau (define (f x_1 x_2 ... x_n) corp)
Perechi și Liste
Operatori condiționali
Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

59

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda, Mașina Markov, Mașina Logică
Paradigme: imperativă, orientată obiect, funcțională, logică
Strategii de evaluare: strictă (ex: aplicativă), nestrictă (ex: normală)
Lambda-expresii: variabilă (x), funcție (λx.e), aplicație ((e_1 e_2))
Sintaxa expresiilor Racket: (funcție e_1 e_2 ... e_n)
Sintaxa funcțiilor Racket: (lambda (x_1 x_2 ... x_n) corp) sau (define (f x_1 x_2 ... x_n) corp)
Perechi și Liste: '(a . b), '(1 2 3), '(), cons, null, list, car, cdr, null?, length, append
Operatori condiționali
Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

60

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda, Mașina Markov, Mașina Logică
Paradigme: imperativă, orientată obiect, funcțională, logică
Strategii de evaluare: strictă (ex: aplicativă), nestrictă (ex: normală)
Lambda-expresii: variabilă (x), funcție (λx.e), aplicație ((e_1 e_2))
Sintaxa expresiilor Racket: (funcție e_1 e_2 ... e_n)
Sintaxa funcțiilor Racket: (lambda (x_1 x_2 ... x_n) corp) sau (define (f x_1 x_2 ... x_n) corp)
Perechi și Liste: '(a . b), '(1 2 3), '(), cons, null, list, car, cdr, null?, length, append
Operatori condiționali: if, cond
Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni

61

Rezumat

Modele de calculabilitate: Mașina Turing, Calculul Lambda, Mașina Markov, Mașina Logică
Paradigme: imperativă, orientată obiect, funcțională, logică
Strategii de evaluare: strictă (ex: aplicativă), nestrictă (ex: normală)
Lambda-expresii: variabilă (x), funcție (λx.e), aplicație ((e_1 e_2))
Sintaxa expresiilor Racket: (funcție e_1 e_2 ... e_n)
Sintaxa funcțiilor Racket: (lambda (x_1 x_2 ... x_n) corp) sau (define (f x_1 x_2 ... x_n) corp)
Perechi și Liste: '(a . b), '(1 2 3), '(), cons, null, list, car, cdr, null?, length, append
Operatori condiționali: if, cond
Soluții pentru înlocuirea atribuirilor, ciclărilor, secvenței de instrucțiuni: compunere de funcții recursive cu starea problemei pasată ca parametru în aceste funcții

62