

# PARADIGME DE PROGRAMARE

## Curs 3

Funcții ca valori de ordinul întâi. Funcționale. Abstractizare.

1

## Funcții ca valori de ordinul întâi – Cuprins

- Importanța Calculului Lambda în matematică și programare
- Valori de ordinul întâi
- Funcții ca valori ale unor variabile / membri ai unor structuri
- Funcții ca valori de retur (funcții curry)
- Funcții ca argumente pentru alte funcții
- Abstractizare
- Abstractizarea la nivel de proces (funcționale)
- Abstractizarea la nivel de date

2

2

## Calcul Lambda

### Istoric

- Inventat de Alonzo Church în 1932 ca un formalism matematic menit să descrie comportamentul computațional al funcțiilor (completând definiția matematică a funcțiilor ca mulțimi de perechi argument-valoare)
- Nu a reușit să înlocuiască teoria mulțimilor ca fundament al matematicii, însă s-a dovedit un model de calculabilitate echivalent cu modelul propus de Turing (Mașina Turing)
- Definiția lui Turing a avut un impact mai mare întrucât propunea un model de mașină pe care să se execute algoritmi
- Privit ca **primul limbaj funcțional**, pe care se bazează toate celelalte

### Aspecte remarcabile

- **Simplitate:** orice valoare se poate modela cu doar 3 constructori
- **Generalitate:** funcțiile se pot aplica pe ele însele (imposibil în teoria mulțimilor, ideea de mulțime care se conține pe ea însăși ducând la paradoxuri)

3

3

## Funcții ca valori de ordinul întâi – Cuprins

- Importanța Calculului Lambda în matematică și programare
- Valori de ordinul întâi
- Funcții ca valori ale unor variabile / membri ai unor structuri
- Funcții ca valori de retur (funcții curry)
- Funcții ca argumente pentru alte funcții
- Abstractizare
- Abstractizarea la nivel de proces (funcționale)
- Abstractizarea la nivel de date

4

4

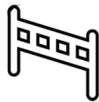
## Valori de ordinul întâi

**Date de ordinul întâi** – pot fi:

- Valori ale unor variabile
- Membri în structuri compuse
- Trimise ca argumente unor funcții
- Valori returnate de o funcție

### Exemple

Lucruri  
Substantive  
Date



### Nu neapărat de ordinul întâi

Acțiuni  
Verbe  
Funcții

5

5

## Valori de ordinul întâi

**Date de ordinul întâi** – în programarea funcțională, avem **doar valori de ordinul întâi!**

- Valori ale unor variabile
- Membri în structuri compuse
- Trimise ca argumente unor funcții
- Valori returnate de o funcție

### Exemple

Lucruri  
Substantive  
Date



### Nu neapărat de ordinul întâi

Acțiuni  
Verbe  
Funcții

6

6

## Funcții ca valori de ordinul întâi – Cuprins

- Importanța Calculului Lambda în matematică și programare
- Valori de ordinul întâi
- **Funcții ca valori ale unor variabile / membri ai unor structuri**
- Funcții ca valori de retur (funcții curry)
- Funcții ca argumente pentru alte funcții
- Abstractizare
- Abstractizarea la nivel de proces (funcționale)
- Abstractizarea la nivel de date

7

7

## Funcții ca valori evaluate la ele însele

1. `(define a +)` ← identificatorul a se leagă la valoarea funcției +
2. `(a 2 3 5)`
- 3.
4. `null?` ← funcția null? se evaluează la ea însăși
- 5.
6. `(define fs (list + (λ (x y) (- x y y)) 5))`
7. `fs` ← al doilea element al listei fs este o funcție anonimă
- 8.
9. `((cadr fs) 20 8)`

8

8

## Funcții ca valori evaluate la ele însele

```

1. (define a +)
2. (a 2 3 5)                ;; 10
3.
4. null?                    ;; #<procedure:null?>
5.
6. (define fs (list + (λ (x y) (- x y y)) 5))
7. fs                      ;; '(#<procedure:+> #<procedure> 5)
8.
9. ((cadr fs) 20 8)        ;; 4

```

9

9

## Funcții ca valori de ordinul întâi – Cuprins

- Importanța Calculului Lambda în matematică și programare
- Valori de ordinul întâi
- Funcții ca valori ale unor variabile / membri ai unor structuri
- Funcții ca valori de retur (funcții curry)
- Funcții ca argumente pentru alte funcții
- Abstractizare
- Abstractizarea la nivel de proces (funcționale)
- Abstractizarea la nivel de date

10

10

## Funcții ca valori de retur

```

1. (define (f x)             ← funcția f (aplicată pe un argument) returnează o funcție
2.   (if (< x 100) + -))     de bibliotecă (+ sau -)
3. (f 25)
4. ((f 120) 16 4)
5.
6. (define (g x)           ← funcția g (aplicată pe un argument) returnează o funcție
7.   (λ (y)                anonimă
8.     (cons x y)))
9. (g 2)
10. ((g 2) '(5 2))

```

11

11

## Funcții ca valori de retur

```

1. (define (f x)
2.   (if (< x 100) + -))
3. (f 25)                ;; #<procedure:+>
4. ((f 120) 16 4)      ;; 12
5.
6. (define (g x)
7.   (λ (y)
8.     (cons x y)))
9. (g 2)                ;; #<procedure>
10. ((g 2) '(5 2))    ;; '(2 5 2)

```

12

12

## Funcții curry / uncurry

În Calculul Lambda există doar funcții **unare**, **anonime** – suficiente pentru a simula orice funcție n-ară

- O funcție binară se poate simula printr-o funcție unară care întoarce o altă funcție unară
- O funcție ternară se poate simula printr-o funcție unară care întoarce o funcție ca mai sus
- ... etc.

### Funcția curry

- **și primește argumentele pe rând**
- Poate fi aplicată parțial (doar pe o parte din argumente) caz în care întoarce o nouă funcție

### Funcția uncurry

- **și primește obligatoriu toate argumentele deodată**
- Nu poate fi aplicată parțial (o asemenea încercare rezultă într-o eroare)

13

13

## Funcții curry / uncurry - Exemple

```

1. (define (plus-curry x)
2.   (λ (y)
3.     (+ x y)))
4.
5. (plus-curry 2)
6. ((plus-curry 2) 10)

7. (define inc (plus-curry 1))
9. (inc 10)

(define (plus-uncurry x y)
  (+ x y))

(plus-uncurry 2 3)

(plus-uncurry 2)

```

Diagram illustrating the flow of arguments:

- Line 2: `(λ (y))` is labeled "per rând" (per line).
- Line 6: `((plus-curry 2) 10)` is labeled "deodată" (all at once).

14

14

## Funcții curry / uncurry - Exemple

```

1. (define (plus-curry x)           (define (plus-uncurry x y)
2.   (λ (y)                         (+ x y))
3.     (+ x y)))
4.
5. (plus-curry 2) ;; #<procedure>   (plus-uncurry 2 3) ;; 5
6. ((plus-curry 2) 10) ;; 12
7.
8. (define inc (plus-curry 1))      (plus-uncurry 2)
9. (inc 10) ;; 11                  ;; plus-uncurry: arity mismatch;

```

15

15

## Funcții curry / uncurry

Conduc la **reutilizare de cod**:

- Permite derivarea facilă de funcții din alte funcții (ex: inc din plus-curry)
- Aceste derivări pot avea loc ad-hoc, acolo unde este nevoie de ele
- **Exemplu** rezolvat la calculator: sortarea prin inserție, folosind un comparator oarecare

16

16

## Funcții ca valori de ordinul întâi – Cuprins

- Importanța Calculului Lambda în matematică și programare
- Valori de ordinul întâi
- Funcții ca valori ale unor variabile / membri ai unor structuri
- Funcții ca valori de retur (funcții curry)
- **Funcții ca argumente pentru alte funcții**
- Abstractizare
- Abstractizarea la nivel de proces (funcționale)
- Abstractizarea la nivel de date

17

17

## Funcții ca argumente pentru alte funcții

### Exemple la calculator:

- Sortarea prin inserție cu comparator oarecare
- O funcție care aplică o transformare oarecare tuturor numerelor pare dintr-o listă
- Similar pentru numere impare, observând un șablon comun și **abstractizând** funcția astfel încât să se poată ușor folosi atât pentru numere pare cât și pentru numere impare

### Observație

- Abstractizarea funcției de mai sus înseamnă o **generalizare**: de la posibilitatea de a transforma numere pare sau impare am generalizat la posibilitatea de a transforma elemente care satisfac o anumită condiție (conceptual, ne-am ridicat la un **nivel mai înalt**, mulțumită observării șablonului comun)

18

18

## Funcții ca valori de ordinul întâi – Cuprins

- Importanța Calculului Lambda în matematică și programare
- Valori de ordinul întâi
- Funcții ca valori ale unor variabile / membri ai unor structuri
- Funcții ca valori de retur (funcții curry)
- Funcții ca argumente pentru alte funcții
- **Abstractizare**
- Abstractizarea la nivel de proces (funcționale)
- Abstractizarea la nivel de date

19

19

## Programele sunt scrise pentru a fi înțelese

### Scop

- Gestiunea complexității intelectuale a programelor (care trebuie întreținute și dezvoltate de oameni, nu doar executate pe niște mașini)

### Mijloace

- **Primitive**
  - datele și funcțiile oferite de limbaj
- **Mijloace de combinare**
  - cum se pot pune primitivele împreună și construi obiecte mai complexe din ele
- **Mijloace de abstractizare**
  - cum se pot folosi combinațiile de elemente primitive ca și când ele însele ar fi primitive

20

20

## Funcții ca valori de ordinul întâi – Cuprins

- Importanța Calculului Lambda în matematică și programare
- Valori de ordinul întâi
- Funcții ca valori ale unor variabile / membri ai unor structuri
- Funcții ca valori de retur (funcții curry)
- Funcții ca argumente pentru alte funcții
- Abstractizare
- Abstractizarea la nivel de proces (funcționale)
- Abstractizarea la nivel de date

21

21

## Abstracțiuni procedurale

- Problemele se descompun în mod natural în **subprobleme**
- Fiecare din aceste subprobleme poate fi rezolvată de o **funcție separată**, care este ca o **cutie neagră** care îndeplinește o sarcină
  - din punct de vedere al problemei principale **nu interesează implementarea** funcției ajutoare (care se poate realiza în diverse moduri)
  - orice funcție capabilă să îndeplinească sarcina este la fel de bună
  - funcția principală se folosește de o **abstracțiune procedurală** mai degrabă decât de o funcție ajutoare concretă (rezolvarea este descrisă **conceptual** mai degrabă decât intrând în detalii de implementare)
- spunem că programarea funcțională este de tip **wishful thinking**
  - descriem soluția în termeni conceptuali, „dorind” ca aceste concepte să existe în limbaj
  - oferim implementări pentru fiecare abstracțiune procedurală folosită de funcția principală

22

22

## Abstracțiuni procedurale - Exemplu

**Exemplu:** primul număr mai mare sau egal decât start care este palindrom în minim b baze dintre bazele 2, 3, 4, 5, 6, 7, 8, 9, 10.

```

1. (define (first-b-pal start b)
2.   (if (min-b-bases? start b '(2 3 4 5 6 7 8 9 10))
3.       start
4.       (first-b-pal (+ 1 start) b)))
5.
6. (define (min-b-bases? n b Bases)
7.   (cond ((zero? b) #t)
8.         ((null? Bases) #f)
9.         ((palindrome? (num->base n (car Bases))) (min-b-bases? n (- b 1) (cdr Bases)))
10.        (else (min-b-bases? n b (cdr Bases)))))

```

wishful thinking: îmi imaginez că min-b-bases? există

apoi o implementez (se putea implementa și altfel, de exemplu numărând toate bazele)

23

23

## Abstracțiuni procedurale – Observații

- O abstracțiune este cu atât mai interesantă cu cât are mai **multe utilizări**
- **Nume sugestive** conduc la programe expresive (apropie programarea de modul în care oamenii gândesc)
- Pentru a folosi o funcție (un feature), utilizatorul trebuie să știe doar **ce face funcția, nu cum** este ea implementată
  - Nu știm cum sunt implementate funcțiile de bibliotecă, le folosim ca pe niște primitive utile (ex: cons, +, equal?)
  - Similar, un dezvoltator al programelor noastre ar trebui să poată folosi funcțiile scrise de noi ca pe niște primitive

24

24

## Recunoașterea unui șablon comun

### Exemple la calculator:

- Adunarea numerelor naturale de la a la b
- Aproximarea lui  $e$  conform seriei  $e = 1/0! + 1/1! + 1/2! + 1/3! + \dots$
- Aproximarea lui  $\pi^2/8$  conform seriei  $\pi^2/8 = 1/1^2 + 1/3^2 + 1/5^2 + \dots$

### Observații

- Funcțiile au foarte mult cod în comun
- Este util să identificăm un șablon mai abstract din care derivă toate cele 3 funcții
- Pași de urmat: **Recunoaștere șablon** → **Definire** → **Reutilizare**

25

25

## Recunoaștere → Definire → Reutilizare

### Recunoașterea șablonului

- Reproducem porțiunile de cod comune
- Porțiunile care diferă devin variabile
- Ex: șablonul comun pentru  $\lambda(2\ 2)$ ,  $\lambda(3\ 3)$ ,  $\lambda(4\ 4)$  este  $(\text{list } x\ x)$

### Definire

- Șablonului identificat la pasul anterior i se atribuie un nume sugestiv, care face programul ușor de înțeles

### Reutilizare

- Cu cât șablonul se întâlnește mai frecvent, cu atât este mai important ca el să fie abstractizat (și multiplele sale instanțe să poată fi astfel derivate cu ușurință)

26

26

## Funcționale (funcții de nivel înalt)

### Funcționale (numite și funcții de nivel înalt)

- **Funcții care primesc ca argumente sau returnează funcții**
- **map, filter, foldl, foldr, apply** – funcționale predefinite în Racket, care abstractizează cele mai comune procese de calcul

### Exemple la calculator:

- Maximul elementelor dintr-o listă de numere
- Extragerea inițialelor dintr-o listă de nume
- Extragerea pronumelor dintr-o listă de cuvinte
- Pătratele elementelor dintr-o listă de numere
- Numărul de elemente pare dintr-o listă
- Extragerea numerelor unei liste care sunt mai mici decât o valoare dată

27

27

## Funcționale (funcții de nivel înalt)

### Funcționale (numite și funcții de nivel înalt)

- **Funcții care primesc ca argumente sau returnează funcții**
- **map, filter, foldl, foldr, apply** – funcționale predefinite în Racket, care abstractizează cele mai comune procese de calcul pe liste

### Exemple la calculator:

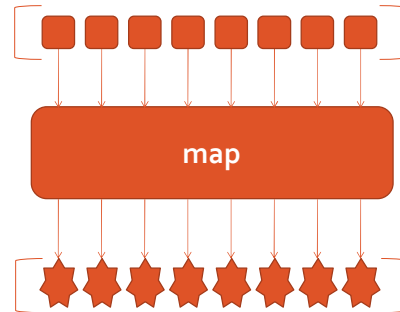
- Maximul elementelor dintr-o listă de numere
- Extragerea inițialelor dintr-o listă de nume
- Extragerea pronumelor dintr-o listă de cuvinte
- Pătratele elementelor dintr-o listă de numere
- Numărul de elemente pare dintr-o listă
- Extragerea numerelor unei liste care sunt mai mici decât o valoare dată

28

28

## Funcționala map (map f L)

```
(map add1 (range 2 7))
(map sqr (range 2 7))
(map even? (range 2 7))
(map random (range 2 7))
```

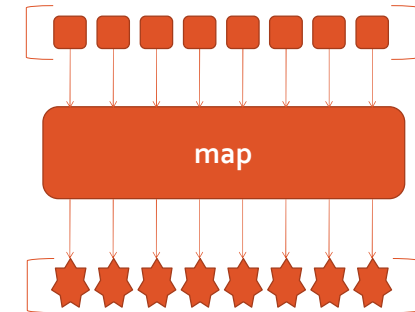


29

29

## Funcționala map (map f L)

```
(map add1 (range 2 7))
;; '(3 4 5 6 7)
(map sqr (range 2 7))
;; '(4 9 16 25 36)
(map even? (range 2 7))
;; '(#t #f #t #f #t)
(map random (range 2 7))
;; surpriză ☺
```

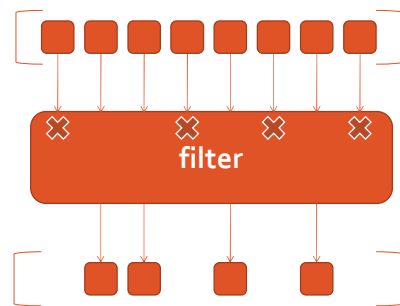


30

30

## Funcționala filter (filter p L)

```
(filter even? (range 2 7))
```

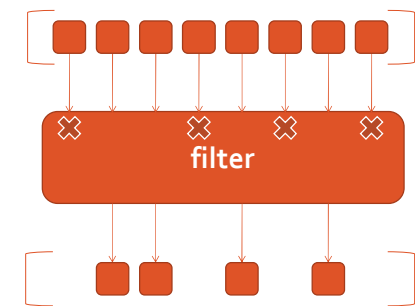


31

31

## Funcționala filter (filter p L)

```
(filter even? (range 2 7))
;; '(2 4 6)
```



32

32

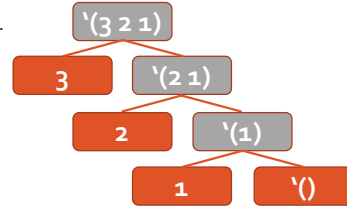


## Funcționala foldl (fold left) (foldl f acc L)

- Folosește o **funcție binară element-acumulator**
- Elementele listei sunt prelucrate în ordinea **stânga→dreapta**
- Mai întâi se aplică funcția pe (first L) și acumulator, rezultând un nou acumulator
- Apoi pe (second L) și noul acumulator ... etc.

### Exemplu

```
(foldl cons '()' '(1 2 3))
```



33

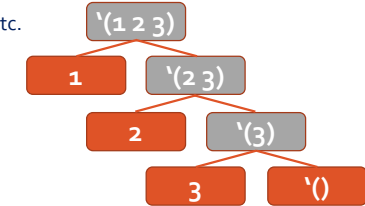
33

## Funcționala foldr (fold right) (foldr f acc L)

- Folosește o **funcție binară element-acumulator**
- Elementele listei sunt prelucrate în ordinea **dreapta→stânga**
- Mai întâi se aplică funcția pe (last L) și acumulator, rezultând un nou acumulator
- Apoi pe penultimul și noul acumulator ... etc.

### Exemplu

```
(foldr cons '()' '(1 2 3))
```



34

34

## Funcționala foldl / foldr

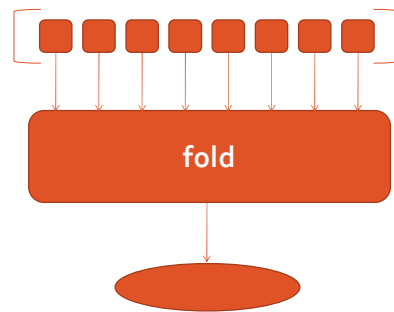
```
(foldl + 0 (range 2 7))
```

```
(foldr max 0 (range 2 7))
```

```
(foldl cons '()' (range 2 7))
```

```
(foldr cons '()' (range 2 7))
```

```
(foldl list 1 (range 2 7))
```



35

35

## Funcționala foldl / foldr

```
(foldl + 0 (range 2 7))
```

```
;; 20
```

```
(foldr max 0 (range 2 7))
```

```
;; 6
```

```
(foldl cons '()' (range 2 7))
```

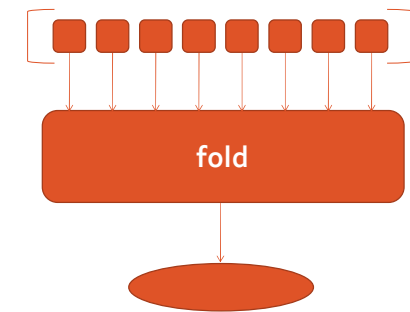
```
;; '(6 5 4 3 2)
```

```
(foldr cons '()' (range 2 7))
```

```
;; '(2 3 4 5 6)
```

```
(foldl list 1 (range 2 7))
```

```
;; '(6 (5 (4 (3 (2 1))))))
```



36

36

## Funcționala apply (apply f [...] L)

(**apply f L**) aplică funcția f pe argumentele din lista L

(**apply f x y z L**) aplică funcția f pe argumentele x y z și cele venite din lista L ... etc.

### Exemple

```
(apply + (range 2 7))      (+ 2 3 4 5 6)
(apply list -1 0 '(1 2 3)) (list -1 0 1 2 3)
(apply map list '((1 2 3))) (map list '(1 2 3))
```

37

37

## Funcționala apply (apply f [...] L)

(**apply f L**) aplică funcția f pe argumentele din lista L

(**apply f x y z L**) aplică funcția f pe argumentele x y z și cele venite din lista L ... etc.

### Exemple

```
(apply + (range 2 7))      ;; 20
(apply list -1 0 '(1 2 3)) ;; '(-1 0 1 2 3)
(apply map list '((1 2 3))) ;; '((1) (2) (3))
```

38

38

## Comparație

(**map f L**) – aplică o transformare f pe **fiecare element** din lista L

(**filter p L**) – păstrează **doar elementele** listei L care satisfac condiția (predicatul) p

(**foldl/foldr f seed L**) – cumulează aplicările funcției f pe **toate elementele** listei L

(**apply f [...] L**) – aplică funcția f pe argumentele conținute în lista L (opțional există și alte argumente date unul câte unul înainte de cele care vor fi despachetate din L)

39

39

## Observație

- Funcționalele de tip fold surprind procesul cel mai general
- **map** și **filter** pot fi implementate cu **fold**
- Pentru expresivitate, este de dorit să folosim
  - **map** când avem de transformat fiecare element (cuvânt cheie: **fiecare**)
  - **filter** când avem de selectat anumite elemente (cuvânt cheie: **selecție**)
- și nu **fold** peste tot.

40

40

## Funcții ca valori de ordinul întâi – Cuprins

- Importanța Calculului Lambda în matematică și programare
- Valori de ordinul întâi
- Funcții ca valori ale unor variabile / membri ai unor structuri
- Funcții ca valori de retur (funcții curry)
- Funcții ca argumente pentru alte funcții
- Abstractizare
- Abstractizarea la nivel de proces (funcționale)
- **Abstractizarea la nivel de date**

41

41

## Tipuri de date abstracte (TDA)

- Ca și procesele frecvente, tipurile de date frecvent utilizate ar trebui să fie abstractizate
- Un TDA se caracterizează prin **constructori** și **operatori**, care se comportă în felul așteptat de utilizator (care nu se mai preocupă de implementarea lor internă)
- Limbajul pune la dispoziție o serie de tipuri primitive (ex: numere, perechi, liste) pe care le manipulăm exclusiv prin constructori și operatori
- Tipurile primitive sunt **pietre de cărămidă pentru construcția de tipuri mai complexe**
- Tipurile mai complexe definite de programator sunt pietre de cărămidă pentru eventuale tipuri (concepte) și mai complexe ... etc.
- Astfel **controlăm complexitatea** intelectuală a unui program mare

42

42

## Abstractizarea la nivel de date

- De fiecare dată când lucrăm cu date compuse (din „cărămizi” mai mici) este bine ca aceste date să fie abstracte
  - În sensul că utilizarea TDA-ului este complet separată (prin ceea ce numim **bariera de abstractizare**) de implementarea sa
- Se definește o **interfață** (un set de constructori și operatori) astfel încât orice manipulare a valorilor TDA-ului să se poată exprima în termeni de acești constructori și operatori
- Programele care manipulează aceste date vor funcționa identic în cazul în care implementarea constructorilor și operatorilor se modifică

### Exemplu

- Numere complexe

```

Utilizare numere complexe
make-complex real imag add-c sub-c mul-c div-c abs-c
Implementare numere complexe (ca perechi, dar irelevant la nivel superior)
cons car cdr
Implementare perechi (nu o știm și nu ne interesează)
  
```

43

43

## Exemplu – Tipul BST (Binary Search Tree)

### Constructori

```
empty-bst : -> BST
make-bst : BST x Elem x BST -> BST
```

### Operatori

```
left : BST -> BST
right : BST -> BST
key : BST -> Elem
bst-empty? : BST -> Bool
insert-bst : Elem x BST -> BST
list->bst : List -> BST
```

44

44

## Rezumat

Date de ordinul întâi  
 Funcții curry / uncurry  
 Funcționale  
 Utilizare map  
 Utilizare filter  
 Utilizare fold  
 Utilizare apply  
 Abstractizare la nivel de proces  
 Abstractizare la nivel de date

45

45

## Rezumat

**Date de ordinul întâi:** valori ale unor variabile, membri în structuri, argumente, valori de retur  
 Funcții curry / uncurry  
 Funcționale  
 Utilizare map  
 Utilizare filter  
 Utilizare fold  
 Utilizare apply  
 Abstractizare la nivel de proces  
 Abstractizare la nivel de date

46

46

## Rezumat

**Date de ordinul întâi:** valori ale unor variabile, membri în structuri, argumente, valori de retur  
**Funcții curry / uncurry:** își primesc argumentele pe rând / își primesc argumentele deodată  
 Funcționale  
 Utilizare map  
 Utilizare filter  
 Utilizare fold  
 Utilizare apply  
 Abstractizare la nivel de proces  
 Abstractizare la nivel de date

47

47

## Rezumat

**Date de ordinul întâi:** valori ale unor variabile, membri în structuri, argumente, valori de retur  
**Funcții curry / uncurry:** își primesc argumentele pe rând / își primesc argumentele deodată  
**Funcționale:** funcții care primesc ca argumente sau returnează funcții  
 Utilizare map  
 Utilizare filter  
 Utilizare fold  
 Utilizare apply  
 Abstractizare la nivel de proces  
 Abstractizare la nivel de date

48

48

## Rezumat

**Date de ordinul întâi:** valori ale unor variabile, membri în structuri, argumente, valori de retur

**Funcții curry / uncurry:** își primesc argumentele pe rând / își primesc argumentele deodată

**Funcționale:** funcții care primesc ca argumente sau returnează funcții

**Utilizare map:** (map f L) aplică f pe fiecare element din L

Utilizare filter

Utilizare fold

Utilizare apply

Abstractizare la nivel de proces

Abstractizare la nivel de date

49

49

## Rezumat

**Date de ordinul întâi:** valori ale unor variabile, membri în structuri, argumente, valori de retur

**Funcții curry / uncurry:** își primesc argumentele pe rând / își primesc argumentele deodată

**Funcționale:** funcții care primesc ca argumente sau returnează funcții

**Utilizare map:** (map f L) aplică f pe fiecare element din L

**Utilizare filter:** (filter p L) păstrează doar elementele din L care satisfac predicatul p

Utilizare fold

Utilizare apply

Abstractizare la nivel de proces

Abstractizare la nivel de date

50

50

## Rezumat

**Date de ordinul întâi:** valori ale unor variabile, membri în structuri, argumente, valori de retur

**Funcții curry / uncurry:** își primesc argumentele pe rând / își primesc argumentele deodată

**Funcționale:** funcții care primesc ca argumente sau returnează funcții

**Utilizare map:** (map f L) aplică f pe fiecare element din L

**Utilizare filter:** (filter p L) păstrează doar elementele din L care satisfac predicatul p

**Utilizare fold:** (foldl/foldr f acc L) parcurge L de la stânga/dreapta, aplicând (f x acc), (f y acc)...

Utilizare apply

Abstractizare la nivel de proces

Abstractizare la nivel de date

51

51

## Rezumat

**Date de ordinul întâi:** valori ale unor variabile, membri în structuri, argumente, valori de retur

**Funcții curry / uncurry:** își primesc argumentele pe rând / își primesc argumentele deodată

**Funcționale:** funcții care primesc ca argumente sau returnează funcții

**Utilizare map:** (map f L) aplică f pe fiecare element din L

**Utilizare filter:** (filter p L) păstrează doar elementele din L care satisfac predicatul p

**Utilizare fold:** (foldl/foldr f acc L) parcurge L de la stânga/dreapta, aplicând (f x acc), (f y acc)...

**Utilizare apply:** (apply f [...] L) aplică f pe argumentele care urmează, despachetând lista L

Abstractizare la nivel de proces

Abstractizare la nivel de date

52

52

## Rezumat

**Date de ordinul întâi:** valori ale unor variabile, membri în structuri, argumente, valori de retur

**Funcții curry / uncurry:** își primesc argumentele pe rând / își primesc argumentele deodată

**Funcționale:** funcții care primesc ca argumente sau returnează funcții

**Utilizare map:** (map f L) aplică f pe fiecare element din L

**Utilizare filter:** (filter p L) păstrează doar elementele din L care satisfac predicatul p

**Utilizare fold:** (foldl/foldr f acc L) parcurge L de la stânga/dreapta, aplicând (f x acc), (f y acc')...

**Utilizare apply:** (apply f [...] L) aplică f pe argumentele care urmează, despachetând lista L

**Abstractizare la nivel de proces:** abstracțiuni procedurale, abstractizare șabloane comune

Abstractizare la nivel de date

53

53

## Rezumat

**Date de ordinul întâi:** valori ale unor variabile, membri în structuri, argumente, valori de retur

**Funcții curry / uncurry:** își primesc argumentele pe rând / își primesc argumentele deodată

**Funcționale:** funcții care primesc ca argumente sau returnează funcții

**Utilizare map:** (map f L) aplică f pe fiecare element din L

**Utilizare filter:** (filter p L) păstrează doar elementele din L care satisfac predicatul p

**Utilizare fold:** (foldl/foldr f acc L) parcurge L de la stânga/dreapta, aplicând (f x acc), (f y acc')...

**Utilizare apply:** (apply f [...] L) aplică f pe argumentele care urmează, despachetând lista L

**Abstractizare la nivel de proces:** abstracțiuni procedurale, abstractizare șabloane comune

**Abstractizare la nivel de date:** TDA-uri cu utilizare separată complet de implementare

54

54