# PARADIGME DE PROGRAMARE

## Curs 5

Funcții ca valori de ordinul întâi.

# 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
  - Continuation passing style

# 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 algoritmii
- 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)

# 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
  - Continuation passing style

# Valori de ordinul întâi

Date de ordinul întâi – pot fi:

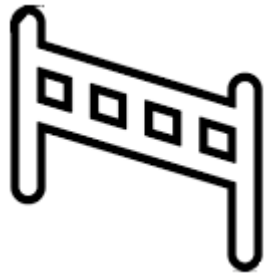
- Valori ale unor variabile
- Membri în structuri compuse
- Trimise ca argumente unor funcții
- Valori returnate de o funcție

Exemple

Lucruri

Substantive

Date



Nu neapărat de ordinul întâi

Acțiuni

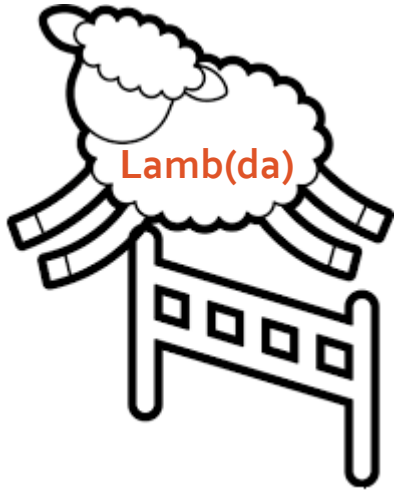
Verbe

Funcții

# 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

# 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
  - Continuation passing style

# Funcții ca valori evaluate la ele însele

```
1.  (define a +)
2.  (a 2 3 5)
3.
4.  null?
5.
6.  (define fs (list + (λ (x y) (- x y y)) 5))
7.  fs
8.
9.  ((cadr fs) 20 8)
```

← identificatorul a se leagă la valoarea funcției +

← funcția null? se evaluează la ea însăși

← al doilea element al listei fs este o funcție anonimă

# 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
```

# 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
  - Continuation passing style

# Funcții ca valori de retur

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

← funcția f (aplicată pe un argument) returnează o funcție de bibliotecă (+ sau -)

← funcția g (aplicată pe un argument) returnează o funcție anonimă

# 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)
```

# 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
  - Continuation passing style

# 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

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

Funcția uncurry

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

# 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.
8.  (define inc (plus-curry 1))
9.  (inc 10)
```

pe rând

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

deodată

```
(plus-uncurry 2 3)

(plus-uncurry 2)
```

# Funcții curry / uncurry - Exemple

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

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


(plus-uncurry 2 3) ;; 5

(plus-uncurry 2)
  ;; plus-uncurry: arity mismatch;
```

# Funcții curry / uncurry – Utilitate

Conduc la reutilizare de cod:

- Permit 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

# 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
  - Continuation passing style

# Funcții ca argumente pentru alte funcții

Exemplu la calculator:

- Sortarea prin inserție
  - Sortare crescătoare
  - Sortare cu comparator oarecare

# 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
  - Continuation passing style

# Exemplu – continuation passing style

```
1.   (define (fact n k)
2.     (if (zero? n)
3.         (k 1)
4.         (fact (- n 1) (compose k (curry * n)))))
5.
6.   (fact 3 identity)
```

```
>(fact 2 (compose identity (curry * 3)))
>(fact 1 (compose identity (curry * 3) (curry * 2)))
>(fact 0 (compose identity (curry * 3) (curry * 2) (curry * 1)))
>((compose identity (curry * 3) (curry * 2) (curry * 1)) 1)
>6
```

# Continuation passing style

## Continuare

- **Funcție care reține restul calculului pentru a îl efectua ulterior**
- La aplicarea continuării k, argumentul său suferă toate prelucrările acumulate în k

## Continuation passing style

- **Stil de programare în care funcțiile nu își întorc direct rezultatul, ci îl pasează ca argument unei continuări**
- Control total asupra modului și momentului în care procesăm rezultatul
  - Rezultatul final nu se mai construiește pe măsură ce avansăm sau revenim din recursivitate
  - Rezultatul final se construiește prin aplicarea continuării asupra unui „punct final” (ex: asupra cazului de bază al recursivității, dar CPS nu se limitează la asta)

# CPS versus recursivitate pe stivă

- CPS este și o tehnică de a transforma recursivitatea pe stivă în recursivitate pe coadă
  - În loc să efectuăm operațiile la revenire, le acumulăm la avans, într-o continuare (ex: fact)
  - Apelul recursiv devine ultima acțiune din funcție, și Racket poate aplica tail-call optimization
- Atenție: Faptul că avem recursivitate pe coadă nu implică eficiență spațială!
  - fact-stack reține O(n) cadre de stivă – pe stivă (pericol: stack overflow)
  - fact-cps reține o funcție care ocupă spațiu O(n) – pe heap (pericol: mult garbage collection)
- În limbajele de programare în general, stiva este mai eficientă ca viteză, iar heap-ul mai rezistent la volum mare de date
- Racket optimizează excelent alocarea pe heap și mecanismul de garbage collection, astfel încât nu se observă diferența de viteză, ci doar rezistența la stack overflow

# Rezumat

Date de ordinul întâi

Funcții curry

Funcții uncurry

Aplicație parțială

Continuare

Continuation passing style

# Rezumat

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

Funcții curry

Funcții uncurry

Aplicație parțială

Continuare

Continuation passing style

# Rezumat

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

Funcții curry: își primesc argumentele pe rând, se pot aplica doar pe o parte din argumente

Funcții uncurry

Aplicație parțială

Continuare

Continuation passing style

# Rezumat

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

Funcții curry: își primesc argumentele pe rând, se pot aplica doar pe o parte din argumente

Funcții uncurry: își primesc argumentele deodată, nu se pot aplica parțial

Aplicație parțială

Continuare

Continuation passing style

# Rezumat

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

Funcții curry: își primesc argumentele pe rând, se pot aplica doar pe o parte din argumente

Funcții uncurry: își primesc argumentele deodată, nu se pot aplica parțial

Aplicație parțială: aplicația unei funcții curry doar pe o parte din argumente, întoarce o nouă funcție care așteaptă restul de argumente

Continuare

Continuation passing style

# Rezumat

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

Funcții curry: își primesc argumentele pe rând, se pot aplica doar pe o parte din argumente

Funcții uncurry: își primesc argumentele deodată, nu se pot aplica parțial

Aplicație parțială: aplicația unei funcții curry doar pe o parte din argumente, întoarce o nouă funcție care așteaptă restul de argumente

Continuare: funcție care reține restul calculului pentru a îl efectua ulterior

Continuation passing style

# Rezumat

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

Funcții curry: își primesc argumentele pe rând, se pot aplica doar pe o parte din argumente

Funcții uncurry: își primesc argumentele deodată, nu se pot aplica parțial

Aplicație parțială: aplicația unei funcții curry doar pe o parte din argumente, întoarce o nouă funcție care așteaptă restul de argumente

Continuare: funcție care reține restul calculului pentru a îl efectua ulterior

Continuation passing style: stil de programare în care funcțiile nu își întorc direct rezultatul, ci îl pasează ca argument unei continuări

- se poate folosi pentru a transforma recursivitatea pe stivă în recursivitate pe coadă (mutând toate operațiile asupra apelurilor recursive într-un parametru de tip continuare)