

# Logică și structuri discrete

## Recursivitate

Casandra Holotescu

casandra@cs.upt.ro

<https://tinyurl.com/lecturesLSD>

Inducție & mulțimi definite inductiv

# Inducție matematică

Dacă o propoziție  $P(n)$  depinde de un număr natural  $n$ , și

1) *cazul de bază* :  $P(0)$  e adevărată

2) *pasul inductiv* : pentru orice  $n \geq 0$

$$P(n) \Rightarrow P(n + 1)$$

atunci  $P(n)$  e adevărată pentru orice  $n$ .

## Mulțimi definite inductiv

Fie mulțimea  $A = \{3, 5, 7, 9, \dots\}$

O putem defini  $A = \{x \mid x = 2k + 3, k \in \mathbb{N}\}$

Alternativ, observăm că:

- ▶  $3 \in A$
- ▶  $x \in A \Rightarrow x + 2 \in A$
- ▶ un element ajunge în  $A$  doar printr-unul din pașii de mai sus

$\Rightarrow$  putem defini *inductiv* mulțimea  $A$

## Mulțimi definite inductiv

$$A = \{3, 5, 7, 9, 11, 13, \dots\}$$

- ▶  $3 \in A$  – **elementul de bază**:  $P(0) : a_0 \in A$
- ▶  $x \in A \Rightarrow x + 2 \in A$  – **construcția de noi elemente**:  
 $P(k) \Rightarrow P(k + 1) : a_k \in A \Rightarrow a_{k+1} \in A$
- ▶ un element ajunge în  $A$  doar printr-unul din pașii de mai sus – **închiderea** (niciun alt element nu e în mulțime)

$\Rightarrow$  definiția *inductivă* a lui  $A$

$\Rightarrow$  spunem că  $A$  e o *mulțime inductivă*

## Mulțimi definite inductiv – formal

O definiție inductivă a unei mulțimi  $S$  constă din:

- ▶ *bază*: Enumerăm **elementele de bază** din  $S$  (minim unul).
- ▶ *inducția*: Dăm cel puțin o **regulă de construcție** de noi elemente din  $S$ , pornind de la elemente deja existente în  $S$ .
- ▶ *închiderea*:  $S$  conține **doar** elementele obținute prin pașii de bază și inducție (de obicei implicită).

Elementele de bază și regulile de construcție de noi elemente constituie **constructorii** mulțimii  $S$ .

## Mulțimi definite inductiv – exemple

Mulțimea numerelor naturale  $\mathbb{N}$  e o mulțime inductivă:

- ▶ *bază*:  $0 \in \mathbb{N}$
- ▶ *inducția*:  $n \in \mathbb{N} \Rightarrow n + 1 \in \mathbb{N}$

Constructorii lui  $\mathbb{N}$ :

baza 0

operația de adunare cu 1

## Mulțimi definite inductiv – exemple

$A = \{1, 3, 7, 15, 31, \dots\}$  e o mulțime inductivă:

▶ *bază*:  $1 \in A$

▶ *inducția*:  $x \in A \Rightarrow 2x + 1 \in A$

Constructorii lui  $A$ :

baza 1

operația de înmulțire cu 2 și adunare cu 1



Recursivitate

# Recursivitate

O noțiune e *recursivă* dacă e *folosită în propria sa definiție*.

Recursivitatea e fundamentală în informatică:

dacă o problemă are soluție, se *poate rezolva recursiv*  
reducând problema la un caz mai simplu al *aceleiași probleme*

⇒ înțelegând recursivitatea, putem rezolva orice problemă  
(dacă e fezabilă)

# Recursivitate: exemple

Recursivitatea

reduce o problemă la *un caz mai simplu* al *aceleiași* probleme

*obiecte*: un *șir* e  $\begin{cases} \text{un singur element} & \text{○} \\ \text{un element urmat de un } \textit{șir} & \text{○} \overbrace{\text{○ ○ ○ ○}}^{\text{șir}} \end{cases}$

ex. cuvânt (șir de litere); număr (șir de cifre zecimale)

*acțiuni*:

un *drum* e  $\begin{cases} \text{un pas} & \longrightarrow \\ \text{un } \textit{drum} \text{ urmat de un pas} & \overbrace{\longrightarrow \longrightarrow \longrightarrow}^{\text{drum}} \longrightarrow \end{cases}$

ex. parcurgerea unei căi într-un graf

## Șiruri recurente

progresie aritmetică:

$$\begin{cases} x_0 = b & (\text{adică: } x_n = b \text{ pentru } n = 0) \\ x_n = x_{n-1} + r & \text{pentru } n > 0 \end{cases}$$

Exemplu: 1, 4, 7, 10, 13, ... ( $b = 1$ ,  $r = 3$ )

progresie geometrică:

$$\begin{cases} x_0 = b & (\text{adică: } x_n = b \text{ pentru } n = 0) \\ x_n = x_{n-1} \cdot r & \text{pentru } n > 0 \end{cases}$$

Exemplu: 3, 6, 12, 24, 48, ... ( $b = 3$ ,  $r = 2$ )

Definițiile de mai sus nu calculează  $x_n$  *direct* ci *din aproape în aproape*, în funcție de  $x_{n-1}$ .

șirul  $x_n$  e *folosit în propria definiție*  $\Rightarrow$  recursivitate / recurență

# Elementele unei definiții recursive

## 1. *Cazul de bază*

= cel mai simplu caz pentru definiția (noțiunea) dată, *definit direct*

termenul inițial dintr-un șir recurent:  $x_0$

un element, în def.: șir = element *sau* șir *urmat de* element

E o *EROARE* dacă lipsește cazul de bază!

## 2. *Relația de recurență* propriu-zisă

– definește noțiunea, folosind *un caz mai simplu* al *aceleiași* noțiuni

3. Demonstrația de *oprire a recursivității* după număr *finit* de pași  
(ex. o mărime nenegativă care descrește când aplicăm definiția)

– la șiruri recurente: *indicele* ( $\geq 0$ , scade în corpul definiției)

– la obiecte: *dimensiunea* (definim obiectul prin alt obiect mai mic)

## Sunt următoarele definiții recursive corecte ?

?  $x_{n+1} = 2 \cdot x_n$

?  $x_n = x_{n+1} - 3$

?  $a^n = a \cdot a \cdot \dots \cdot a$  (de  $n$  ori)

? o frază e o înșiruire de cuvinte

? un șir e un șir mai mic urmat de un alt șir mai mic

? un șir e un caracter urmat de un șir

O definiție recursivă trebuie să fie *bine formată* (v. condițiile 1-3)  
ceva *nu* se poate defini *doar* în funcție de sine însuși  
se pot utiliza doar noțiuni *deja* definite  
nu se poate genera un calcul *infini*t (trebuie să se oprească)

# Funcții recursive

# Funcții recursive

O funcție e *recursivă* dacă apare în propria sa definiție.

O funcție  $f$  e definită recursiv dacă există cel puțin o valoare  $f(x)$  definită în termenii altei valori  $f(y)$ , unde  $x \neq y$ .



# Funcții recursive peste mulțimi inductive

Multe funcții recursive au ca domeniu **mulțimi inductive**.

Dacă  $S$  este o mulțime inductivă, putem folosi **constructorii** săi pentru a defini o funcție recursivă  $f$  cu domeniul  $S$

- ▶ *baza*: pentru **fiecare** element de bază  $x \in S$  specificăm o valoare  $f(x)$
- ▶ *inducția*: dăm una sau mai multe reguli care pentru orice  $x \in S$ ,  $x$  definit inductiv, definesc  $f(x)$  în termenii unei/unor alte valori ale lui  $f$ , **definite anterior**

## Funcții recursive – exemple

Să definim recursiv funcția

$$f : \mathbb{N} \rightarrow \mathbb{N}, f(n) = 1 + 3 + \dots + (2n + 1)$$

$\mathbb{N}$  e o mulțime inductivă:

- ▶ *bază*:  $0 \in \mathbb{N}$
- ▶ *inducția*:  $n \in \mathbb{N} \Rightarrow n + 1 \in \mathbb{N}$

Definiția recursivă a lui  $f$ :

- ▶ *bază*:  $\mathbf{f(0) = 1}$
- ▶ *inducția*:  
 $f(n + 1) = 1 + 3 + \dots + (2n + 1) + (2(n + 1) + 1)$   
 $f(n + 1) = 1 + 3 + \dots + (2n + 1) + (2n + 3)$   
 $\mathbf{f(n + 1) = f(n) + (2n + 3)}$  pt.  $n > 0$

## Funcții recursive – exemple

Să definim recursiv funcția factorial

$$\text{factorial} : \mathbb{N} \rightarrow \mathbb{N}, \text{factorial}(n) = 1 * 2 * 3 * \dots * (n - 1) * n$$

$\mathbb{N}$  e o mulțime inductivă:

- ▶ *bază*:  $0 \in \mathbb{N}$
- ▶ *inducția*:  $n \in \mathbb{N} \Rightarrow n + 1 \in \mathbb{N}$

Definiția recursivă a lui *factorial*:

- ▶ *bază*: **factorial(0) = 1**
- ▶ *inducția*:  $\text{factorial}(n + 1) = 1 * 2 * 3 * \dots * (n - 1) * n * (n + 1)$   
**factorial(n + 1) = factorial(n) \* (n + 1)** pt.  $n > 0$

Să programăm funcții recursive!

# Recapitulare

Am definit funcții într-un *limbaj de programare funcțional*.

*Domeniul* și *codomeniul* sunt *tipuri* în limbajele de programare.

Tipurile ne spun pe ce fel de valori poate fi folosită o funcție

`let comp f g x = f (g x)`

`val comp: ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>`

`g` are tipul `'c -> 'a` și `f` are tipul `'a -> 'b`

⇒ domeniul de valori al lui `g` e domeniul de definiție al lui `f`

compunerea are tipul `'c -> 'b` 'a, 'b, 'c pot fi orice tip

Funcțiile pot avea ca *argumente* și/sau *rezultat* alte *funcții*

*Compunând* funcții ( $f \circ g$ ) *rezolvăm probleme* mai complexe:

`g` produce un rezultat, `f` îl folosește mai departe

## Ce putem face până acum

Putem defini funcții simple:

```
let max x y = if x > y then x else y
```

e de fapt predefinită, nu e nevoie s-o definim încă o dată

Putem compune de un număr dat de ori (număr fix de argumente)

```
let max3 x1 x2 x3 = max x1 (max x2 x3)
```

```
let max4 x1 x2 x3 x4 = max x1 (max x2 (max x3 x4))
```

Nu putem încă:

exprima că vrem să lucrăm cu **N** valori (listă, mulțime, tablou)  
defini un calcul pentru un număr **arbitrar** de valori

## Progresia ca funcție recursivă

$$\text{Progresia aritmetică: } \begin{cases} x_0 = b \\ x_n = x_{n-1} + r \quad \text{pentru } n > 0 \end{cases}$$

Fie o progresie aritmetică cu baza și rația fixate:

$$x_0 = 3, x_n = x_{n-1} + 2 \quad (\text{pentru } n > 0)$$

Noțiunea recursivă (șirul) devine o *funcție*

Valoarea de care depinde (indicele) devine *argumentul* funcției

```
let rec ap3r2 n =  
  if n = 0 then 3  
  else 2 + ap3r2 (n-1)
```

## Funcții recursive în ML

```
let rec ap3r2 n =  
  if n = 0 then 3  
  else 2 + ap3r2 (n-1)
```

Cuvintele cheie **let rec** introduc o *definiție recursivă*:

funcția *ap3r2* e folosită (apelată) în propria definiție

Fără **rec**, fie *ap3r2* din dreapta ar fi necunoscut (eroare), fie s-ar folosi o eventuală definiție anterioară (deci nu ar fi recursivă).



# Mecanismul apelului recursiv

Fiecare apel face “*în cascadă*” *un nou apel*, până la cazul de bază

Fiecare apel execută *același cod*, dar cu *alte date*  
(valori proprii pentru parametri)

Ajunși la cazul de bază, toate apelurile făcute sunt încă *neterminate*  
(fiecare mai are de făcut adunarea cu rezultatul apelului efectuat)

Revenirea se face *în ordine inversă* apelării  
(ultimul apelul revine primul, apoi revine penultimul apel, etc.)

În interpretor, putem vizualiza apelurile și revenirea cu directiva  
`#trace numefuncție`  
revenim la normal cu `#untrace numefuncție`

## Potrivirea de tipare

Putem scrie funcția și așa:

```
let rec ap3r2 indice = match indice with  
  | 0 -> 3  
  | n -> 2 + ap3r2 (n-1)
```

`ap3r2` e o funcție:

- dacă argumentul e 0, valoarea funcției e 3
- dacă argumentul are orice altă valoare (o notăm  $n$ ), valoarea funcției e  $2 + \text{ap3r2 } (n-1)$

`match indice with`

definește *potrivire de tipare*, cu parametrul `indice`

Fiecare ramură definește în stânga lui `->` un *tipar* și în dreapta *rezultatul* (putem folosi numele introduse în tiparul din stânga)

## Potrivirea de tipare

Sau așa, tot cu potrivire de tipare:

```
let rec ap3r2 = function  
  | 0 -> 3  
  | n -> 2 + ap3r2 (n-1)
```

La fel ca `match` arg `with`, dar fără argument explicit

Cuvântul cheie `function` introduce un nou argument implicit și face *potrivire de tipare* după acesta

## Potrivirea de tipare (cont.)

```
let rec ap3r2 indice = match indice with
| 0 -> 3
| n -> 2 + ap3r2 (n-1)
```

Argumentul care e potrivit cu tiparul poate fi:

- o *constantă* (aici, 0)
- o *valoare structurată* (pereche, listă cu cap/coadă, etc.)
  - perechile se notează  $(x, y)$  ca în matematică
  - triplele:  $(a, b, c)$ , etc
- un *identificator* (nume) care indică tot argumentul (oricare ar fi)

Nu putem avea ca tipar (doar) o condiție  ~~$x \rightarrow 5$~~

Potrivirile se încearcă în ordinea indicată, până la prima reușită.

## Potrivirea de tipare (cont.)

Identificatorul special `_` (linie de subliniere) *se potrivește cu **orice*** `Îl folosim dacă nu avem nevoie de valoarea respectivă.`

Dacă am uitat un tipar posibil, compilatorul ne avertizează.

```
let pozitie coord = match coord with
  | (0, 0) -> print_string "origine"
  | (_, 0) -> print_string "pe axa x"
  | (0, y) -> Printf.printf "pe axa y la %d" y
  | (_, _) -> print_string "nu e pe axe"
```

## Potrivirea de tipare: exemple

Ex.: o funcție care ia triplete de întregi și dă suma componentelor până la primul zero.

```
let sumto0 t = match t with
  | (0, _, _) -> 0
  | (x, 0, _) -> x
  | (x, y, z) -> x + y + z
```

dacă prima componentă e 0, rezultatul e 0, *indiferent de celelalte*  
altfel, dacă a doua componentă e 0, adunăm *doar prima* (nu și a treia)  
altfel, primele două sunt nenule, și le însumăm pe toate trei

Rescriere echivalentă cu if-then-else:

```
let sumto0 (x, y, z) =
  if x = 0 then 0 else if y = 0 then x else x + y + z
```

## Definiții locale

Până acum: definiții *globale*

```
let identificador = expresie
```

```
let fct arg1 ... argN = expresie
```

Uneori sunt utile *definiții auxiliare*. Am vrea să scriem:

Definim funcția *arie*( $a, b, c$ ) astfel: ( $a, b, c$  = laturile unui triunghi)

întâi definim  $p = (a + b + c)/2$

cu această notație, aria e  $\sqrt{p(p-a)(p-b)(p-c)}$

```
let arie a b c = (* traducem in ML *)
```

```
  let p = (a +. b +. c) /. 2. in
```

```
    sqrt (p *. (p -. a) *. (p -. b) *. (p -. c))
```

## Sintaxă: Definiții locale

```
let arie a b c = (* traducem in ML *)  
  let p = (a +. b +. c) /. 2. in  
  sqrt (p *. (p -. a) *. (p -. b) *. (p -. c))
```

Definiția e tot de forma

```
let funcție arg1 arg2 ... argN = expresie
```

dar **expresie** are noua formă:

```
let id_aux = expr_aux in expr_val
```

Funcția are valoarea lui **expr\_val**,  $\sqrt{p(p-a)(p-b)(p-c)}$ , unde *id\_aux* (adică *p*) are sensul  $p = (a+b+c)/2$ .

Astfel dăm un nume, *p*, pentru o expresie folosită de mai multe ori,  $(a+b+c)/2$ , scriind mai concis și evitând recalcularea.



## Exemplu: generalizăm progresia aritmetică

Putem scrie o funcție care are *baza* și *rația* ca parametri:

```
let rec ap baza ratie indice = match indice with  
  | 0 -> baza  
  | n -> ratie + ap baza ratie (n-1)
```

sau echivalent

```
let rec ap baza ratie n =  
  if n = 0 then baza else ratie + ap baza ratie (n-1)
```

Putem defini apoi funcții care corespund unor progresii individuale:

```
let ap3r2 = ap 3 2      (* baza 3, ratia 2 *)  
# ap3r2 4  
- : int = 11           (* termenul de indice 4 *)
```

## Rescriem cu definiții locale

```
let rec ap baza ratie indice = match indice with  
  | 0 -> baza  
  | n -> ratie + ap baza ratie (n-1)
```

Apare de două ori expresia `ap baza ratie`, o funcție de un argument (`indice`), în care `baza` și `ratie` sunt deja fixate.

Rescriem dând un nume `ap1` pentru *expresia comună*.

```
let ap baza ratie =  
  let rec ap1 indice = match indice with  
    | 0 -> baza  
    | n -> ratie + ap1 (n-1)  
  in ap1
```

## Rescriem cu definiții locale

Rescriem dând un nume `ap1`  
pentru *expresia comună*  
(definiție locală pentru `ap1`)

În exterior definim funcția inițială  
`ap` baza rație egală cu `ap1`

```
let ap baza rație =  
  let rec ap1 = function  
    | 0 -> baza  
    | n -> rație + ap1 (n-1)  
  in ap1
```

Citim: `let` (fie) `ap` baza rație definită astfel:

- definim funcția `ap1` (folosind parametrii lui `ap`: baza, rație;  
 `ap1` ia ca arg. indicele `n` și dă valoarea termenului al `n`-lea)
- atunci `ap` baza rație e chiar `ap1` (expresia de după `in`)

`ap1` are rol ajutător, nu e vizibil în afara definiției lui `ap`

## Alt exemplu clasic: “problema $3 \cdot n + 1$ ”

Fie un număr pozitiv  $n$ :

dacă e par, îl împărțim la 2:  $n/2$

dacă e impar, îl înmulțim cu 3 și adunăm 1:  $3 \cdot n + 1$

$$f(n) = \begin{cases} n/2 & \text{dacă } n \text{ par} \\ 3 \cdot n + 1 & \text{altfel} \end{cases}$$

Se ajunge la 1 pornind de la orice număr pozitiv ?

(problemă nerezolvată...)

= Conjectura lui Collatz (1937), cunoscută sub multe alte nume

Exemple:

$3 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$

$11 \rightarrow 34 \rightarrow 17 \rightarrow 52 \rightarrow 26 \rightarrow 13 \rightarrow 40 \rightarrow 20 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$

## Câți pași până la oprire?

Definim funcția  $p : \mathbb{N}^* \rightarrow \mathbb{N}$  care numără pașii până la oprire:  
pentru  $3 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$  avem 7 pași

Nu avem o formulă cu care să definim  $p(n)$  direct.

Dar dacă șirul  $n, f(n), f(f(n)), \dots$  ajunge la 1,  
atunci numărul de pași parcurși **de la**  $n$   
e **cu unul mai mare** decât continuând **de la**  $f(n)$

$$p(n) = \begin{cases} 0 & \text{dacă } n = 1 \text{ (am ajuns)} \\ 1 + p(f(n)) & \text{altfel (dacă } n > 1) \end{cases}$$

Funcția  $p$  e folosită în propria definiție, deci a fost definită *recursiv*.

```
let f n = if n mod 2 = 0 then n / 2 else 3 * n + 1
```

```
let rec p n = if n = 1 then 0 else 1 + p (f n)
```

# Recursivitate structurală

## Calculul expresiilor aritmetice

O *expresie* (puțin mai) complicată:

$$(2 + 3) * (4 + 2 * 3) - 5 * 6 / (7 - 2) + (4 + 3 - 2) / (7 - 3)$$

Pentru a calcula, trebuie să înțelegem *structura* expresiei

E *suma* a două subexpresii (+ e calculat ultimul):

$$\begin{aligned} & (2 + 3) * (4 + 2 * 3) - 5 * 6 / (7 - 2) \\ + & (4 + 3 - 2) / (7 - 3) \end{aligned}$$

Apoi calculăm *expresiile mai simple*

$$(2 + 3) * (4 + 2 * 3) - 5 * 6 / (7 - 2) = 44$$

$$(4 + 3 - 2) / (7 - 3) = 1$$

$$44 + 1 = 45$$

Calculul celor două subexpresii: după *aceleași reguli*

# Pași în rezolvarea problemei

Ce ne-a permis să calculăm expresia complicată?

- ▶ *Identificarea structurii recursive*  
*expresia* e suma a două *expresii* mai simple  
vom folosi *tipuri de date* definite *recursiv*
- ▶ Exprimăm *pașii de calcul* elementari (cei mai simpli)  
putem aduna, împărți, etc. două *numere*
- ▶ Identificăm *condiția de oprire*  
când expresia e un simplu număr, nu mai trebuie făcut nimic



## Expresia ca noțiune recursivă

Ce e o expresie aritmetică?

int + int      5 + 2

int - int      2 - 3

int \* int      -1 \* 4

int / int      7 / 3

Se poate mai simplu? Da: int      (5 e caz particular de expresie)

Se poate și mai complicat? Da:

int \* (int + int)

(int - int) / (int \* int)

...

Putem scrie un număr finit de reguli ?

## Expresia, definită recursiv

O *expresie*:  $\left\{ \begin{array}{l} \text{întreg} \\ \text{expresie} + \text{expresie} \\ \text{expresie} - \text{expresie} \\ \text{expresie} * \text{expresie} \\ \text{expresie} / \text{expresie} \end{array} \right.$

Am descris expresia printr-o *gramatică* (niște reguli de scriere):  
așa se descriu limbajele de programare

detalii despre gramatici într-un alt curs

urmăriți *diagramele de sintaxă* la cursul de programare

*selection-statement:*

```
if ( expression ) statement
```

```
if ( expression ) statement else statement
```

```
switch ( expression ) statement
```

# Recursivitate structurală în ML

## Tipuri recursive

Pentru a reprezenta structura recursivă a unei probleme, ne trebuie adesea *date* definite recursiv. În ML putem *construi* tipuri recursive.

Un *tip recursiv* pentru expresii (incluzând operatorii de calcul):

```
type expr = I of int
          | Add of expr * expr
          | Sub of expr * expr
          | Mul of expr * expr
          | Div of expr * expr
```

Am definit un tip cu mai multe *variante*.

Tipul `expr` e *recursiv* (o valoare de tip expresie poate conține la rândul ei componente de tip expresie)

## Tipuri recursive

```
type expr = I of int
          | Add of expr * expr | Sub of expr * expr
          | Mul of expr * expr | Div of expr * expr
```

Am definit un tip cu mai multe *variante*.

Fiecare variantă trebuie scrisă cu un *constructor de tip* (etichetă), ales de noi: I, Add, etc. (orice identificator cu literă mare)

Notăția  $\text{expr} * \text{expr}$  reprezintă *produsul cartezian*, deci o pereche de două valori de tipul  $\text{expr}$

Expresia  $(2 + 3) * 7$  se reprezintă:  $\text{Mul} (\text{Add}(\text{I } 2, \text{I } 3), \text{I } 7)$

## Evaluarea recursivă a unei expresii

Lucrul cu o valoare de tip recursiv se face prin *potrivire de tipare* (engl. pattern matching), *pentru fiecare variantă* din tip

```
let rec eval = function
| I i -> i
| Add (e1, e2) -> eval e1 + eval e2
| Sub (e1, e2) -> eval e1 - eval e2
| Mul (e1, e2) -> eval e1 * eval e2
| Div (e1, e2) -> eval e1 / eval e2
```

Evaluând expresia `eval (Mul (Add(I 2, I 3), I 7))` dă 35.  
e nevoie de paranteze, pentru a grupa Mul și perechea de după

Pentru *tipuri* definite *recursiv*

*funcțiile* care îl prelucrează vor fi natural *recursive*

de obicei cu câte un caz pentru fiecare variantă a tipului respectiv

# De știut

Să recunoaștem și definim *noțiuni recursive*

Să recunoaștem dacă o definiție recursivă e *corectă*  
(are caz de bază? se oprește recursivitatea?)

Să rezolvăm probleme scriind *funcții* recursive  
cazul de bază + pasul de reducere la o problemă mai simplă

Să definim și folosim *tipuri recursive*