# Paradigme de Programare

Mihnea Muraru
mihnea.muraru@upb.ro

2025–2026, semestrul 2

# Partea I

## Introducere

# Cuprins



# Cuprins



# Notare

- Curs: 0,5p
- Laborator: 0,2p
- Teste: 1,3p sau 1,8p
  (0,3p grile săptămânale + 1p sau 1,5p final)
- Teme: 2,5p sau 3p
  (2 teme, 8 etape săptămânale în total)
- Examen (*open-book*): 5p
- Bonus-uri la testele săptămânale, teme și examen

# Regulament

Vă rugăm să citiți regulamentul cu atenție!

`https://ocw.cs.pub.ro/courses/pp/26/regulament`

# Desfășurarea cursului

- Recapitularea cursului anterior
  + exercițiu similar subiectelor de examen
- Predare
- Activitate de curs
- Legătură strânsă între curs, laborator
  și etapele temelor!

# Activitatea de curs

- Obținerea de puncte pe:
  - **Întrebări** de profunzime, care să vizeze legături între concepte, studiate inclusiv la alte materii
  - Răspunsuri la întrebările **titularului**
  - Răspunsuri la întrebările **altor colegi**
- La finalul cursului, rezumarea intervențiilor proprii în activitatea de pe **Moodle**

# Cuprins



# Ce vom studia?

1. Modele de calculabilitate:

2. Paradigme de programare:

3. Limbaje de programare:

# Ce vom studia?

1. Modele de calculabilitate:
   Diverse perspective conceptuale asupra noțiunii de calculabilitate efectivă

2. Paradigme de programare:

3. Limbaje de programare:

# Ce vom studia?

1. Modele de calculabilitate:
   Diverse perspective conceptuale asupra noțiunii de calculabilitate efectivă

2. Paradigme de programare:
   Influența perspectivei alese asupra procesului de modelare și rezolvare a problemelor

3. Limbaje de programare:

# Ce vom studia?

1. Modele de calculabilitate:
   Diverse perspective conceptuale asupra noțiunii de calculabilitate efectivă

2. Paradigme de programare:
   Influența perspectivei alese asupra procesului de modelare și rezolvare a problemelor

3. Limbaje de programare:
   Mecanisme expresive, aferente paradigmelor, cu accent pe aspectul comparativ

# De ce?

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

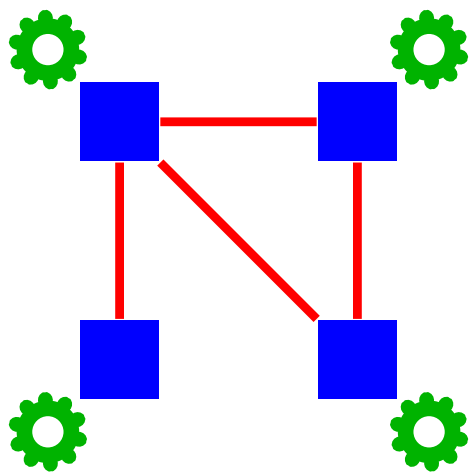
---

Edsger Dijkstra,
*How do we tell truths that might hurt*

# Descompunerea problemelor

Controlul complexității: descompunere și interfațare

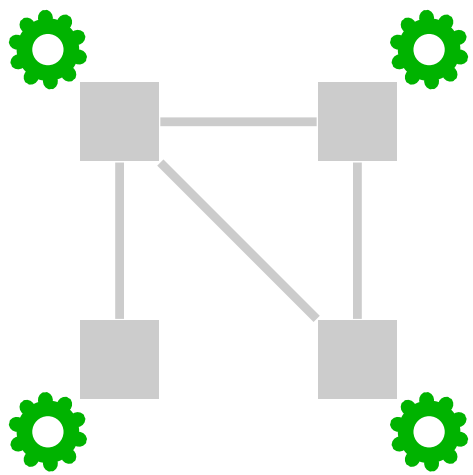
| Descompunere | Accent pe | Rezultat |
|---|---|---|
| | | |
| | | |
| | | |
| | | |



# Descompunerea problemelor

Controlul complexității: descompunere și interfațare

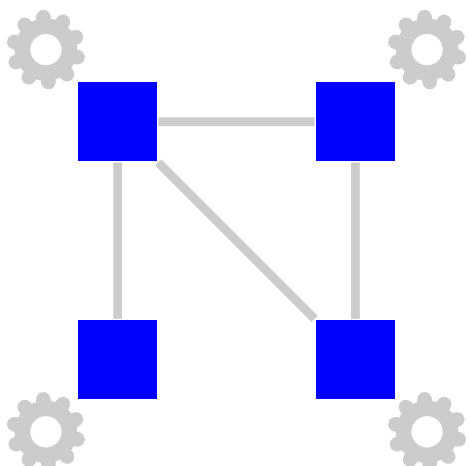
| Descompunere | Accent pe | Rezultat |
|---|---|---|
| Procedurală | Acțiuni | Proceduri |
| | | |
| | | |
| | | |



# Descompunerea problemelor

Controlul complexității: descompunere și interfațare

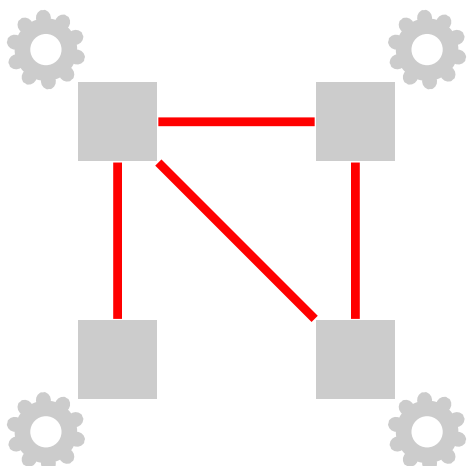
| Descompunere | Accent pe | Rezultat |
|---|---|---|
| Procedurală | Acțiuni | Proceduri |
| Orientată obiect | Entități | Clase și obiecte |
| | | |
| | | |



# Descompunerea problemelor

Controlul complexității: descompunere și interfațare

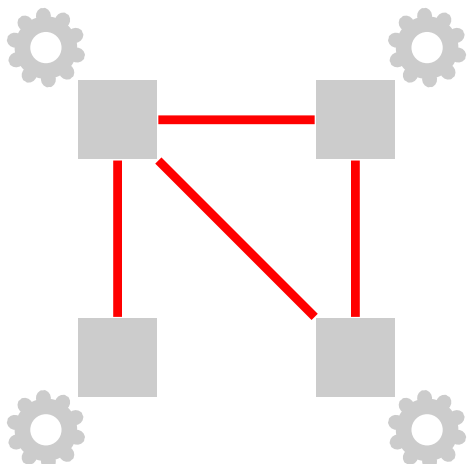
| Descompunere | Accent pe | Rezultat |
|---|---|---|
| Procedurală | Acțiuni | Proceduri |
| Orientată obiect | Entități | Clase și obiecte |
| Funcțională | Relații | Funcții în sens matematic |
| | | |



# Descompunerea problemelor

Controlul complexității: descompunere și interfațare

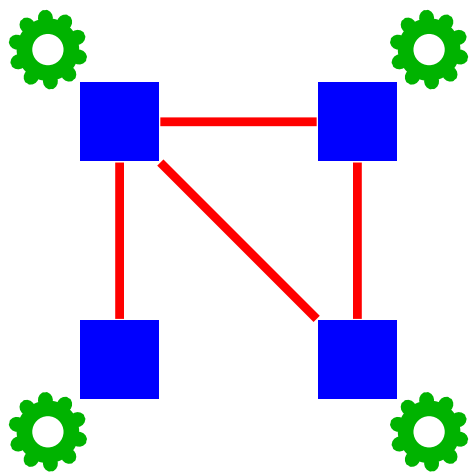
| Descompunere | Accent pe | Rezultat |
|---|---|---|
| Procedurală | Acțiuni | Proceduri |
| Orientată obiect | Entități | Clase și obiecte |
| Funcțională | Relații | Funcții în sens matematic |
| Logică | Relații | Predicate și propoziții |



# Descompunerea problemelor

Controlul complexității: descompunere și interfațare

| Descompunere | Accent pe | Rezultat |
|---|---|---|
| Procedurală | Acțiuni | Proceduri |
| Orientată obiect | Entități | Clase și obiecte |
| Funcțională | Relații | Funcții în sens matematic |
| Logică | Relații | Predicate și propoziții |



# De ce? (cont.)

- Lărgirea spectrului de abordare a problemelor

# De ce? (cont.)

- Lărgirea spectrului de abordare a problemelor
- Identificarea perspectivei ce permite modelarea simplă a unei probleme; alegerea limbajului adecvat

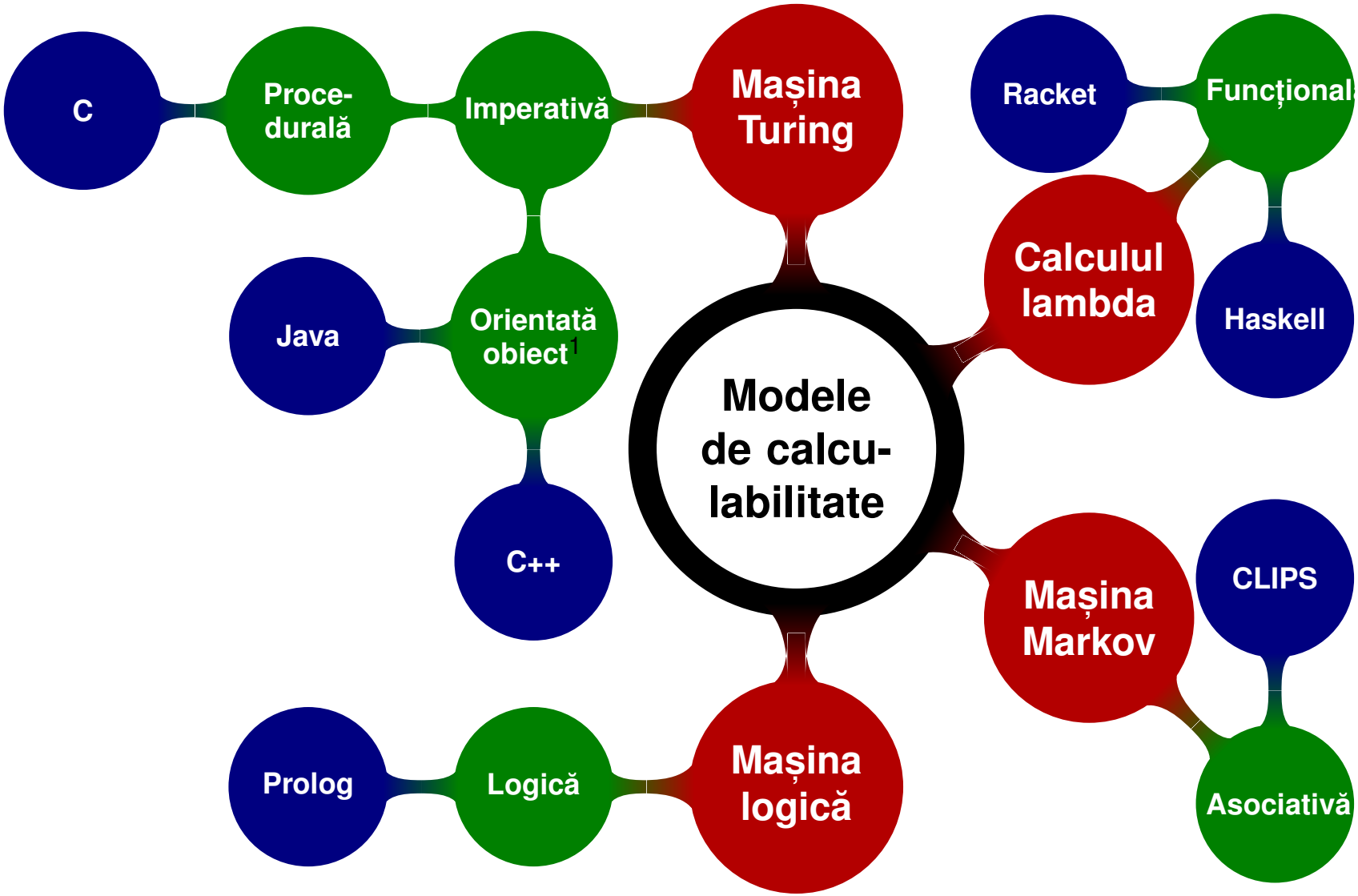
# De ce? (cont.)

- Lărgirea spectrului de abordare a problemelor
- Identificarea perspectivei ce permite modelarea simplă a unei probleme; alegerea limbajului adecvat
- Exploatarea mecanismelor oferite de limbajele de programare (v. Dijkstra!)

# De ce? (cont.)

- Lărgirea spectrului de abordare a problemelor
- Identificarea perspectivei ce permite modelarea simplă a unei probleme; alegerea limbajului adecvat
- Exploatarea mecanismelor oferite de limbajele de programare (v. Dijkstra!)
- Sporirea capacității de învățare a noi limbaje și de adaptare la particularitățile și diferențele dintre acestea

# Modele, paradigme, limbaje




[1] Original imperativă, dar se poate combina chiar cu abordarea funcțională

# Limitele calculabilității

- Teza Church-Turing:
  efectiv calculabil $\equiv$ Turing calculabil

# Limitele calculabilității

- Teza Church-Turing:
  efectiv calculabil $\equiv$ Turing calculabil

- Echivalența celorlalte modele de calculabilitate,
  și a multor altora, cu Mașina Turing

# Limitele calculabilității

- Teza Church-Turing:
  efectiv calculabil $\equiv$ Turing calculabil
- Echivalența celorlalte modele de calculabilitate, și a multor altora, cu Mașina Turing
- Există vreun model superior ca forță de calcul?

# Cuprins



# O primă problemă

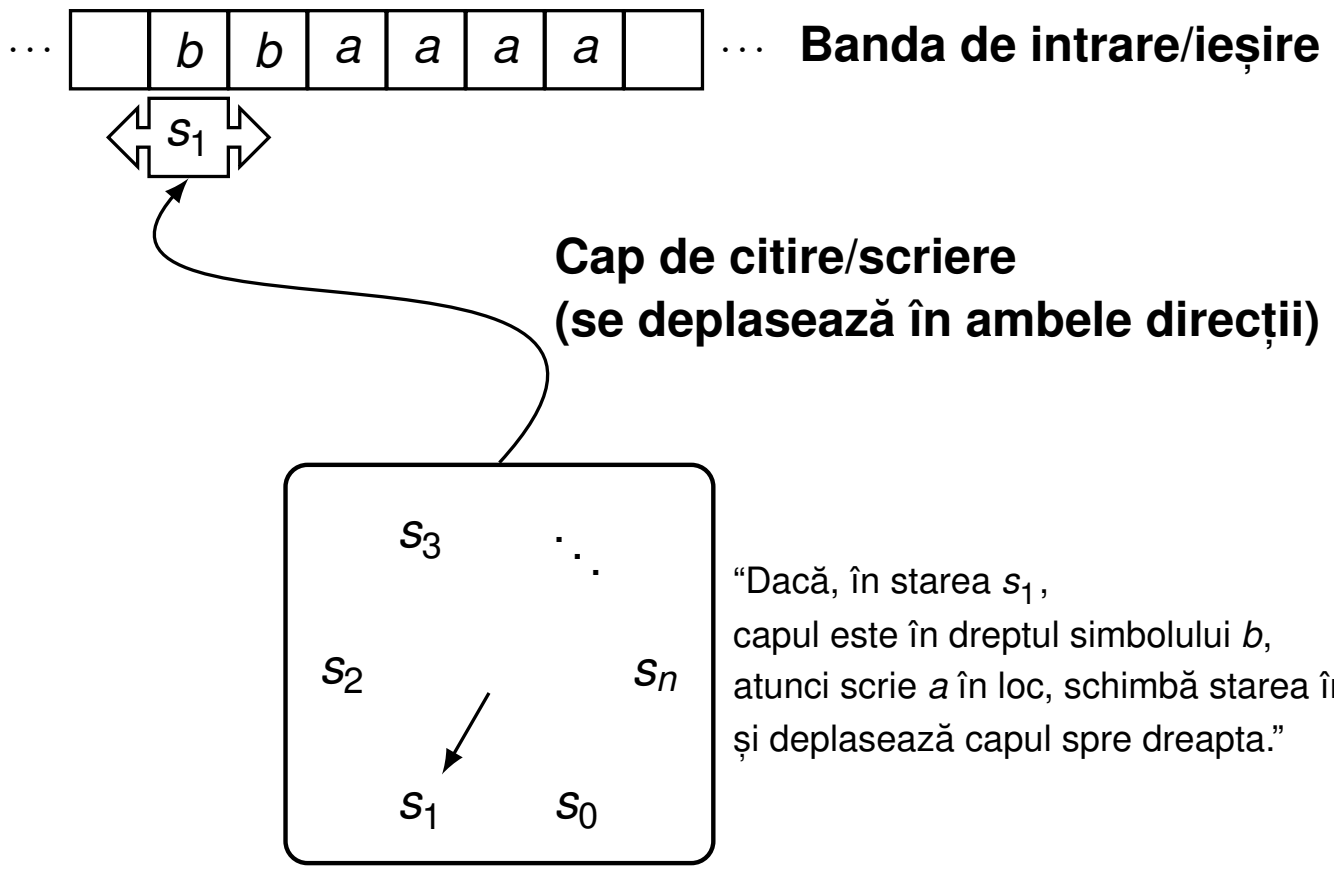
**Example 3.1.**

Să se determine elementul minim dintr-un vector.

# Abordare imperativă

## Modelul

### Mașina Turing



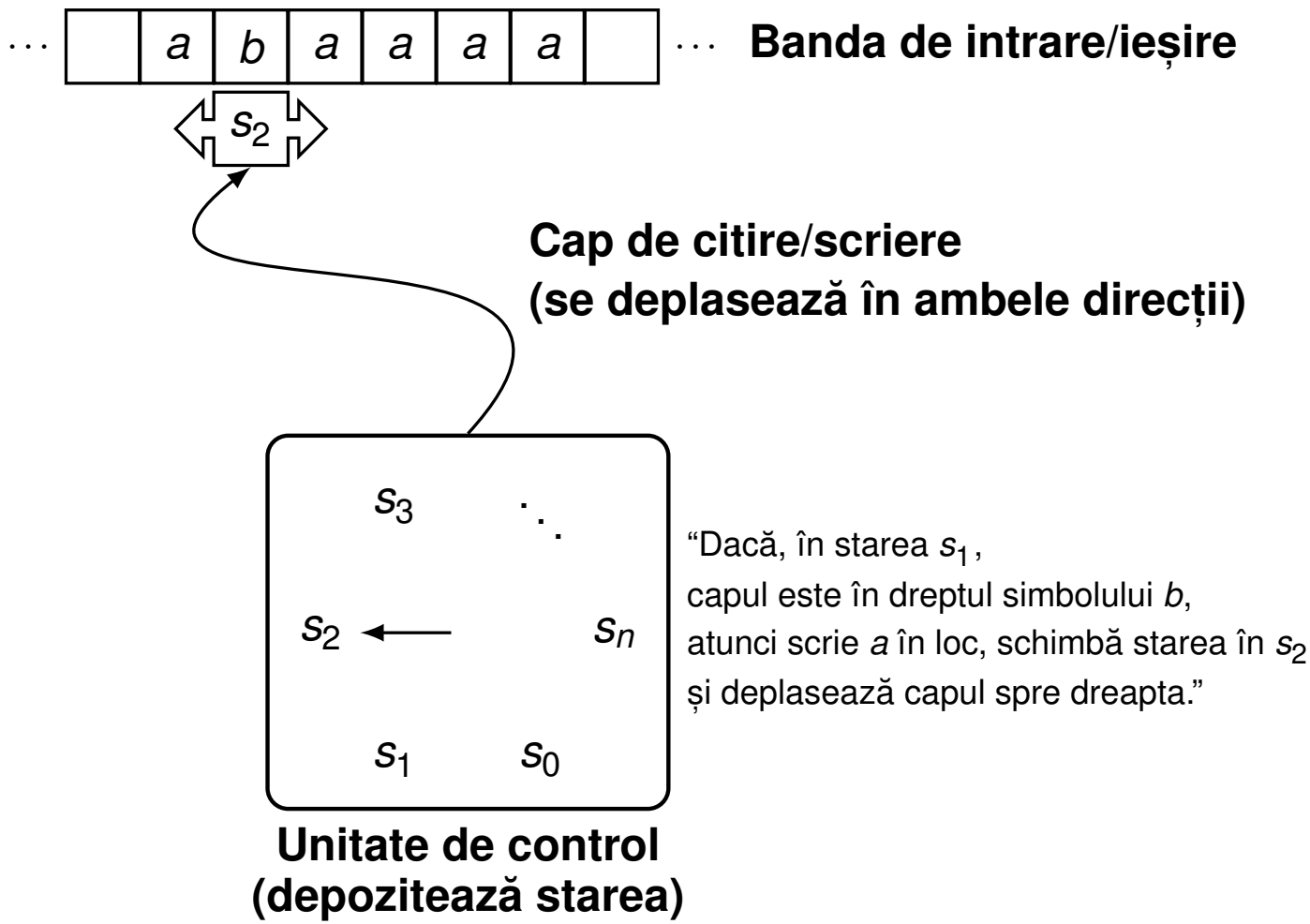

"Dacă, în starea $s_1$,
capul este în dreptul simbolului *b*,
atunci scrie *a* în loc, schimbă starea în $s_2$
și deplasează capul spre dreapta."

Prelucrare după: `http://www.texample.net/tikz/examples/turing-machine-2/`

# Abordare imperativă

## Modelul

### Mașina Turing




"Dacă, în starea $s_1$,
capul este în dreptul simbolului *b*,
atunci scrie *a* în loc, schimbă starea în $s_2$
și deplasează capul spre dreapta."

Prelucrare după: `http://www.texample.net/tikz/examples/turing-machine-2/`

# Abordare imperativă (procedurală)

## Limbajul

```
1:  procedure MinList(L, n)
2:      min ← L[1]
3:      i ← 2
4:      while i ≤ n do
5:          if L[i] < min then
6:              min ← L[i]
7:          end if
8:          i ← i + 1
9:      end while
10:     return min
11: end procedure
```

# Abordare imperativă

## Paradigma

- Orientare spre acțiuni și efectele acestora

# Abordare imperativă

## Paradigma

- Orientare spre acțiuni și efectele acestora
- “Cum” se obține soluția, pașii de urmat

# Abordare imperativă

## Paradigma

- Orientare spre acțiuni și efectele acestora
- “Cum” se obține soluția, pașii de urmat
- Atribuirea ca operație fundamentală

# Abordare imperativă

## Paradigma

- Orientare spre acțiuni și efectele acestora
- “Cum” se obține soluția, pașii de urmat
- Atribuirea ca operație fundamentală
- Programe cu stare

# Abordare imperativă

## Paradigma

- Orientare spre acțiuni și efectele acestora
- “Cum” se obține soluția, pașii de urmat
- Atribuirea ca operație fundamentală
- Programe cu stare
- Secvențierea instrucțiunilor

Calculul lambda

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

“Pentru a aplica funcția $\lambda x.x$

# Abordare funcțională

## Modelul

Calculul lambda

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

"Pentru a aplica funcția $\lambda x.x$ asupra parametrului actual, $y$,

# Abordare funcțională

## Modelul

Calculul lambda

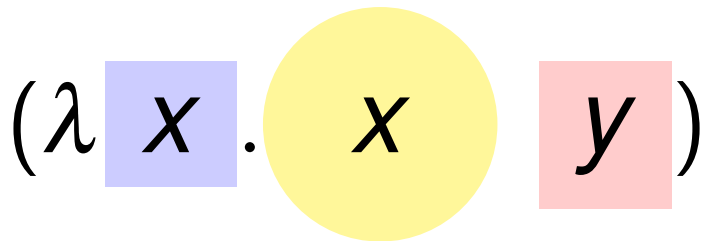
$$(\lambda\ x\ .\quad x\qquad y\ )$$

"Pentru a aplica funcția $\lambda x.x$ asupra parametrului actual, $y$, se indentifică parametrul formal, $x$,

# Abordare funcțională

## Modelul

Calculul lambda

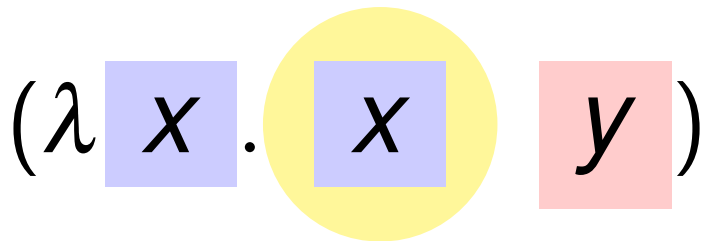



"Pentru a aplica funcția $\lambda x.x$ asupra parametrului actual, $y$, se indentifică parametrul formal, $x$, în corpul funcției, $x$,

# Abordare funcțională

## Modelul

### Calculul lambda

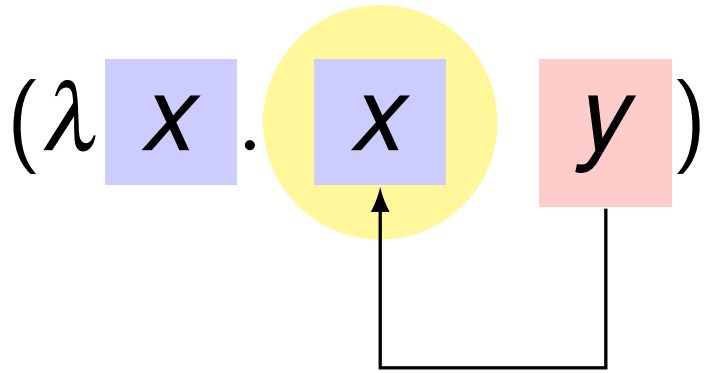



"Pentru a aplica funcția $\lambda x.x$ asupra parametrului actual, $y$, se indentifică parametrul formal, $x$, în corpul funcției, $x$, iar aparițiile primului, $x$ (singura),

# Abordare funcțională

## Modelul

### Calculul lambda

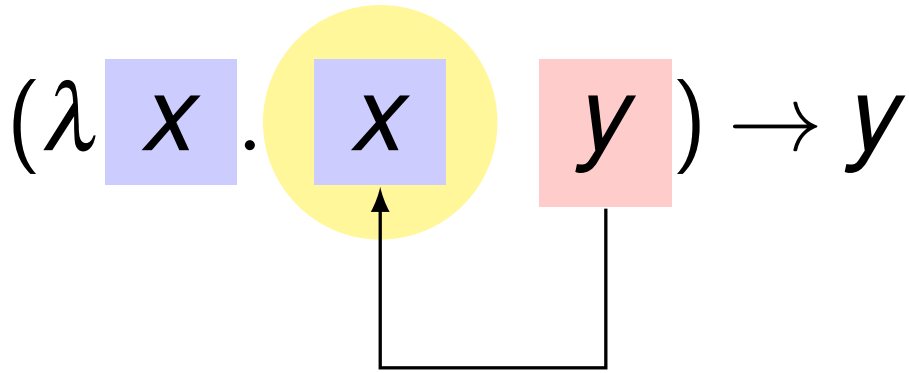



"Pentru a aplica funcția $\lambda x.x$ asupra parametrului actual, $y$, se indentifică parametrul formal, $x$, în corpul funcției, $x$, iar aparițiile primului, $x$ (singura), se **substituie** cu parametrul actual,

# Abordare funcțională

## Modelul

Calculul lambda




"Pentru a aplica funcția $\lambda x.x$ asupra parametrului actual, $y$, se indentifică parametrul formal, $x$, în corpul funcției, $x$, iar aparițiile primului, $x$ (singura), se **substituie** cu parametrul actual, obținându-se rezultatul unui pas de evaluare."

# Abordare funcțională

## Limbajul

- Racket (2 variante):

```
1  (define (minList1 L)
2    (if (null? (cdr L))
3        (car L)
4        (min (car L) (minList1 (cdr L)))))
5
6  (define (minList2 L)
7    (foldl min (car L) (cdr L)))
```

# Abordare funcțională

## Limbajul

- Racket (2 variante):

```
1  (define (minList1 L)
2    (if (null? (cdr L))
3        (car L)
4        (min (car L) (minList1 (cdr L)))))
5
6  (define (minList2 L)
7    (foldl min (car L) (cdr L)))
```

- Haskell (aceleași 2 variante):

```
1  minList1 [h]     = h
2  minList1 (h : t) = min h (minList1 t)
3
4  minList2 (h : t) = foldl min h t
```

# Abordare funcțională

## Paradigma

- Funcții matematice, care transformă intrările în ieșiri

# Abordare funcțională

## Paradigma

- Funcții matematice, care transformă intrările în ieșiri
- Absența atribuirilor și a stării

# Abordare funcțională

## Paradigma

- Funcții matematice, care transformă intrările în ieșiri
- Absența atribuirilor și a stării
- Funcții ca valori de prim rang (e.g., ca parametri ai altor funcții)

# Abordare funcțională

## Paradigma

- Funcții matematice, care transformă intrările în ieșiri
- Absența atribuirilor și a stării
- Funcții ca valori de prim rang (e.g., ca parametri ai altor funcții)
- Recursivitate, în locul iterării

# Abordare funcțională

## Paradigma

- Funcții matematice, care transformă intrările în ieșiri
- Absența atribuirilor și a stării
- Funcții ca valori de prim rang (e.g., ca parametri ai altor funcții)
- Recursivitate, în locul iterării
- Compunere de funcții, în locul secvențierii instrucțiunilor

# Abordare funcțională

## Paradigma

- Funcții matematice, care transformă intrările în ieșiri
- Absența atribuirilor și a stării
- Funcții ca valori de prim rang (e.g., ca parametri ai altor funcții)
- Recursivitate, în locul iterării
- Compunere de funcții, în locul secvențierii instrucțiunilor
- Diminuarea importanței ordinii de evaluare

# Abordare funcțională

## Paradigma

- Funcții matematice, care transformă intrările în ieșiri
- Absența atribuirilor și a stării
- Funcții ca valori de prim rang (e.g., ca parametri ai altor funcții)
- Recursivitate, în locul iterării
- Compunere de funcții, în locul secvențierii instrucțiunilor
- Diminuarea importanței ordinii de evaluare
- Funcții de ordin superior (i.e. care iau alte funcții ca parametru, e.g., `foldl`)

Logica cu predicate de ordin I

$$muritor(Socrate) \quad om(Platon) \quad \forall x.om(x) \Rightarrow muritor(x)$$

Logica cu predicate de ordin I

$$muritor(Socrate) \quad om(Platon) \quad \forall x.om(x) \Rightarrow muritor(x)$$

“La ce se poate lega variabila $y$
astfel încât $muritor(y)$ să fie satisfăcută?”

Logica cu predicate de ordin I

$$muritor(Socrate) \quad om(Platon) \quad \forall x.om(x) \Rightarrow muritor(x)$$

"La ce se poate lega variabila $y$
astfel încât $muritor(y)$ să fie satisfăcută?"

$$y \leftarrow Socrate \quad \text{sau} \quad y \leftarrow Platon$$

# Abordare logică

## Limbajul

- Axiome:

# Abordare logică

## Limbajul

- Axiome:
  1. $x \leq y \Rightarrow min(x, y, x)$

# Abordare logică

## Limbajul

- Axiome:
  1. $x \leq y \Rightarrow min(x, y, x)$
  2. $y < x \Rightarrow min(x, y, y)$

# Abordare logică

## Limbajul

- Axiome:
  1. $x \leq y \Rightarrow min(x, y, x)$
  2. $y < x \Rightarrow min(x, y, y)$
  3. $minList([m], m)$

# Abordare logică

## Limbajul

- Axiome:
  1. $x \leq y \Rightarrow min(x, y, x)$
  2. $y < x \Rightarrow min(x, y, y)$
  3. $minList([m], m)$
  4. $minList([y|t], n) \wedge min(x, n, m) \Rightarrow minList([x, y|t], m)$

# Abordare logică

## Limbajul

- Axiome:
  1. $x \leq y \Rightarrow min(x, y, x)$
  2. $y < x \Rightarrow min(x, y, y)$
  3. $minList([m], m)$
  4. $minList([y|t], n) \wedge min(x, n, m) \Rightarrow minList([x, y|t], m)$

- Prolog:

```
1  min(X, Y, X) :- X =< Y.
2  min(X, Y, Y) :- Y < X.
3
4  minList([M], M).
5  minList([X, Y | T], M) :-
6      minList([Y | T], N), min(X, N, M).
```

# Abordare logică

## Paradigma

- Formularea proprietăților logice ale obiectelor și soluției

# Abordare logică

## Paradigma

- Formularea proprietăților logice ale obiectelor și soluției
- Flux de control implicit, dirijat de date

# Abordările funcțională și logică

## Asemănări

- Formularea proprietăților soluției

# Abordările funcțională și logică

## Asemănări

- Formularea proprietăților soluției
- “Ce” trebuie obținut (vs. “cum” la imperativă)

# Abordările funcțională și logică

## Asemănări

- Formularea proprietăților soluției
- “Ce” trebuie obținut (vs. “cum” la imperativă)
- Se subsumează abordării declarative, opuse celei imperative

# Cuprins



# Ce este o paradigmă de programare?

- Un set de convenții care dirijează maniera în care gândim programele

# Ce este o paradigmă de programare?

- Un set de convenții care dirijează maniera în care gândim programele

- Ea dictează modul în care:

# Ce este o paradigmă de programare?

- Un set de convenții care dirijează maniera în care gândim programele

- Ea dictează modul în care:
  - reprezentăm datele

# Ce este o paradigmă de programare?

- Un set de convenții care dirijează maniera în care gândim programele

- Ea dictează modul în care:
  - reprezentăm datele
  - operațiile prelucrează datele respective

# Ce este o paradigmă de programare?

- Un set de convenții care dirijează maniera în care gândim programele
- Ea dictează modul în care:
  - reprezentăm datele
  - operațiile prelucrează datele respective
- Abordările anterioare reprezintă paradigme de programare (procedurală, funcțională, logică)

# Accepții asupra limbajelor

- Modalitate de exprimare a instrucțiunilor pe care calculatorul le execută

# Accepții asupra limbajelor

- Modalitate de exprimare a instrucțiunilor pe care calculatorul le execută
- Mai important, modalitate de exprimare a unui mod de gândire

# Accepții asupra limbajelor

*. . . "computer science" is not a science and [. . . ] its significance has little to do with computers. The computer revolution is a revolution in the way we think and in the way we express what we think.*

---

Harold Abelson et al.,
*Structure and Interpretation of Computer Programs*

# Câteva trăsături

- Tipare
  - Statică/ dinamică
  - Tare/ slabă

# Câteva trăsături

- Tipare
  - Statică/ dinamică
  - Tare/ slabă
- Ordinea de evaluare a parametrilor funcțiilor
  - Aplicativă
  - Normală

# Câteva trăsături

- Tipare
  - Statică/ dinamică
  - Tare/ slabă
- Ordinea de evaluare a parametrilor funcțiilor
  - Aplicativă
  - Normală
- Legarea variabilelor
  - Statică
  - Dinamică

# Rezumat

Importanța cunoașterii
paradigmelor și limbajelor de programare,
în scopul identificării celor convenabile
pentru modelarea unei probleme particulare

# Partea II

# Tipuri de date abstracte

# Cuprins



# Cuprins



# Motivație (1)

- Separație dezirabilă între **specificația** unui tip de date și **implementările** sale — exemplu în Java:
  - Interfața `List` vs. clasele `ArrayList` și `LinkedList`
  - De interes **ce** face `get(i)` (întoarce elementul de pe poziția `i`), nu **cum** realizează acest lucru
- **Proprietăți** impuse pentru interacțiunea operațiilor:
  - Stabilirea unui element la el însuși nu schimbă lista
  - `list.set(i, list.get(i))` $\equiv$ `list`

# Motivație (2)

- **Demonstrarea** de noi proprietăți pe baza celor impuse, independent de implementare
  - Dubla inversare a unei liste nu schimbă lista
  - `list.reversed().reversed()` ≡ `list`
- Garanția că orice implementare care satisface proprietățile **impuse** satisface și proprietățile **derivate**

# Cuprins



# Definiție

- Tip de date abstract (TDA): model matematic al unei **mulțimi** de valori, împreună cu **operațiile** pe acestea și **proprietățile** aferente
- Exemple: număr natural, listă, stivă, arbore binar, expresie aritmetică etc.
- Componente:
  - Constructori
  - Operatori
  - Axiome

# Componente

- **Constructori**: metode de creare a valorilor tipului
  - Exemplu: crearea unei liste *singleton* dintr-un număr natural
- **Operatori**: metode de interogare a valorilor
  - Exemplu: calculul lungimii unei liste
- **Axiome**: proprietăți care guvernează interacțiunea constructorilor și operatorilor
  - Exemplu: adăugând un element într-o listă, lungimea crește cu 1

# TDA `Natural`

- **Constructori**:
  - `0 : Natural` (*zero*)
  - `S : Natural` $\rightarrow$ `Natural` (*succesor*)
  - `P : Natural \ {0}` $\rightarrow$ `Natural` (*predecesor*)
  - `+ : Natural` $\times$ `Natural` $\rightarrow$ `Natural` (*adunare*)
  - ...
- **Operatori**:
  - `zero? : Natural` $\rightarrow$ `Bool`
  - `even? : Natural` $\rightarrow$ `Bool`
  - ...
- **Axiome**: cum le scriem?

# Scrierea axiomelor

- În principiu, necesitatea explicitării interacțiunilor dintre constructori și operatori
- Problemă: număr posibil prea mare de constructori
- Soluție: izolarea unui număr redus de constructori care împreună pot genera toate valorile tipului
  $\Rightarrow$ **constructori de bază**
- Rezultat: scrierea axiomelor (explicitarea comportamentului) pentru constructorii rămași și operatorii doar în raport cu **constructorii de bază**

# TDA `Natural`

- Constructori de bază:
  - `0 : Natural` (*zero*)
  - `S : Natural` $\rightarrow$ `Natural` (*succesor*)
- Notație: aplicația lui `f` asupra lui `x`, desemnată prin `(f x)`, în loc de `f(x)` (anticipând sintaxa limbajelor funcționale)
- Exemple de expresii:
  - `(S (S 0))`, echivalent cu 2
  - `(P (S (S 0))) + (S 0) = (S (S 0))`
  - `(even? (S (S 0))) = true`

# Axiomele TDA `Natural`

- `P`
  - `(P (S n)) = n` (P.S)
- `+`
  - `0 + n = n` (+.0)
  - `(S m) + n = (S (m + n))` (+.S)
- `zero?`
  - `(zero? 0) = true` (zero?.0)
  - `(zero? (S n)) = false` (zero?.S)
- `even?`
  - `(even? 0) = true` (even?.0)
  - `(even? (S 0)) = false` (even?.S0)
  - `(even? (S (S n))) = (even? n)` (even?.SS)

# TDA `List`

- **Constructori de bază**:
  - `[] : List` (*empty*)
  - `:` `: T × List → List` (*cons*)
- **Alți constructori**:
  - `singleton : T → List`
  - `tail : List \ {[]} → List`
  - `++ : List × List → List` (*append*)
  - `reverse : List → List`
  - `...`
- **Operatori**:
  - `empty? : List → Bool`
  - `head : List \ {[]} → T`
  - `length : List → Natural`
  - `...`

# Axiomele TDA `List` (1)

- `tail`
  - `(tail (x : xs)) = x` (tail.:)
- `++`
  - `[] ++ xs = xs` (++.[])
  - `(x : xs) ++ ys = x : (xs ++ ys)` (++.:)
- `reverse`
  - `(reverse []) = []` (reverse.[])
  - `(reverse (x : xs))`
    `= (reverse xs) ++ (x : [])` (reverse.:)

# Axiomele TDA `List` (2)

- `empty?`
  - `(empty? []) = true` (empty?.[])
  - `(empty? (x : xs)) = false` (empty?.:)
- `head`
  - `(head (x : xs)) = x` (head.:)
- `length`
  - `(length []) = 0` (length.[])
  - `(length (x : xs))`
    `= (S (length xs))` (length.:)

# Clasificarea constructorilor

- **Nulari**: fără parametri
  - `0` pentru `Natural`
  - `[]` pentru `List`
- **Externi**: cu parametri de alt tip decât cel construit
  - `singleton` pentru `List`
- **Interni**: cu parametri de același tip ca cel construit
  - `S` pentru `Natural`
  - `:` pentru `List`

# Dimensiunea valorilor TDA

- Numărul **constructorilor de bază** din descrierea valorii
- Exemple:
  - `(S (S 0))`: dimensiune 3
  - `1 : (2 : (3 : []))`: dimensiune 4
- Corespondență natură constructor – dimensiune:
  - Nular: dimensiune 1
  - Extern: dimensiune 1
  - Intern: dimensiune $> 1$

# Cuprins



# Inducție matematică

- Cum putem demonstra că adunarea numerelor naturale, pentru care am scris deja axiome, este **asociativă** sau **comutativă**?
- **Inducție matematică** pentru demonstrarea proprietății $P$ pentru numere naturale:
  - Cazul de bază: $P(0)$
  - Pasul inductiv: $P(n) \implies P(n+1)$
  - Concluzie: $\forall n.P(n)$
- Corespondența cazurilor de mai sus cu **constructorii de bază** ai TDA `Natural`!
- Posibilitatea generalizării pentru orice TDA, în forma **inducției structurale**

# Inducție structurală

- Necesitatea asumării proprietății pentru fiecare dintre obiectele mai **simple** pe baza cărora este construit obiectul mai **complex** pentru care vrem să demonstrăm aceeași proprietate
- Similară cu inducția matematică, dar realizată după **dimensiunea** valorilor
- **Cazurile de bază**:
  - Pentru fiecare constructor **nular** $c_{\text{nul}}$: $P(c_{\text{nul}})$
  - Pentru fiecare constructor **extern** $c_{\text{ext}}$ cu parametrii $p_i$: $P((c_{\text{ext}}\ p_1\ \ldots\ p_n))$
- **Pașii inductivi**:
  - Pentru fiecare constructor **intern** $c_{\text{int}}$ cu parametrii $p_i$, dintre care $t_j \in \text{TDA}$: $\bigwedge_j P(t_j) \implies P((c_{\text{int}}\ p_1 \ldots\ p_n))$

# Asociativitatea adunării (1)

- Proprietatea Assoc(x):

```
∀ y, z . (x + y) + z = x + (y + z)
```

- Cazul de bază: Assoc(0):

```
(0 + y) + z = 0 + (y + z)
```

```
(0 + y) + z
= {+.0}
y + z
```

```
0 + (y + z)
= {+.0}
y + z
```

# Asociativitatea adunării (2)

- Pasul inductiv: $\text{Assoc}(\texttt{x}) \Longrightarrow \text{Assoc}(\texttt{(S x)})$:

```
((S x) + y) + z = (S x) + (y + z)
```

```
((S x) + y) + z
= {+.s}
(S (x + y)) + z
= {+.s}
(S ((x + y) + z))
= {ipoteza inductivă}
(S (x + (y + z)))
```

```
(S x) + (y + z)
= {+.s}
(S (x + (y + z)))
```

# Comutativitatea adunării (1)

- Proprietatea Commut(x):

  $\forall$ y . x + y = y + x

- Cazul de bază: Commut(0):

  0 + y = y + 0

0 + y

**= {+.0}**

y

y + 0

**= {Lema 1}**

y

# Comutativitatea adunării (2)

- Pasul inductiv: Commut(`x`) $\Longrightarrow$ Commut(`(S x)`):

```
(S x) + y = y + (S x)
```

`(S x) + y`

**= {+.s}**

`(S (x + y))`

**= {ipoteza inductivă}**

`(S (y + x))`

`y + (S x)`

**= {Lema 2}**

`(S (y + x))`

# Lema 1 (1)

- Proprietatea $\mathrm{L1}(\mathtt{y})$:

  $\mathtt{y + 0 = y}$

- Cazul de bază: $\mathrm{L1}(\mathtt{0})$:

  $\mathtt{0 + 0 = 0}$

  $\underline{\mathtt{0 + 0}}$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $\mathtt{0}$

  $\mathtt{= \{+.0\}}$

  $\mathtt{0}$

# Lema 1 (2)

- Pasul inductiv: $\mathrm{L1}(\mathtt{y}) \Longrightarrow \mathrm{L1}(\mathtt{(S\ y)})$:

  `(S y) + 0 = (S y)`

  `(S y) + 0` `(S y)`

  **= {+.s}**

  `(S (y + 0))`

  **= {ipoteza inductivă}**

  `(S y)`

# Lema 2 (1)

- Proprietatea L2(y):

  ∀ x . y + (S x) = (S (y + x))

- Cazul de bază: L2(0):

  0 + (S x) = (S (0 + x))

0 + (S x)

**= {+.0}**

(S x)

(S (0 + x))

**= {+.0}**

(S x)

# Lema 2 (2)

- Pasul inductiv: $\mathrm{L2}(\texttt{y}) \Longrightarrow \mathrm{L2}(\texttt{(S y)})$:

```
(S y) + (S x) = (S ((S y) + x))
```

```
(S y) + (S x)
= {+.s}
(S (y + (S x)))
= {ipoteza inductivă}
(S (S (y + x)))
```

```
(S ((S y) + x))
= {+.s}
(S (S (y + x)))
```

# Observații

- În anumite cazuri, necesitatea demonstrării unor proprietăți ajutătoare (lemele 1 și 2)

- Posibilitatea schimbării variabilei în raport cu care se realizează inducția ($\mathtt{x}$, respectiv $\mathtt{y}$)

# Inversarea listelor ca involuție (1)

- Proprietatea RevInv(`xs`):

```
(reverse (reverse xs)) = xs
```

- Cazul de bază: RevInv(`[]`):

```
(reverse (reverse [])) = []
```

```
(reverse (reverse []))                    []
= {rev.[]}
(reverse [])
= {rev.[]}
[]
```

# Inversarea listelor ca involuție (2)

- Pasul inductiv: RevInv(`xs`) $\Longrightarrow$ RevInv(`x : xs`):

```
(reverse (reverse (x : xs))) = x : xs
```

```
(reverse (reverse (x : xs)))
```

**= {rev.:}**

```
(reverse ((reverse xs) ++ (x : [])))
```

**= {Lema 3}**

```
(reverse (x : [])) ++ (reverse (reverse xs))
```

**= {ipoteza inductivă}**

```
(reverse (x : [])) ++ xs
```

# Inversarea listelor ca involuție (3)

```
= {rev.:}
((reverse []) ++ (x : [])) ++ xs
= {rev.[]}
([] ++ (x : [])) ++ xs
= {++.[]}
(x : []) ++ xs
= {++.:}
x : ([] ++ xs)
= {++.[]}
x : xs
```

# Lema 3

- Proprietatea $\mathrm{L3}(\texttt{xs})$:

```
∀ ys . (reverse (xs ++ ys))
    = (reverse ys) ++ (reverse xs)
```

- Cazul de bază: $\mathrm{L3}(\texttt{[]})$:

```
(reverse ([] ++ ys))
    = (reverse ys) ++ (reverse [])
```

- Pasul inductiv: $\mathrm{L3}(\texttt{xs}) \Longrightarrow \mathrm{L3}(\texttt{x : xs})$:

```
(reverse ((x : xs) ++ ys))
    = (reverse ys) ++ (reverse (x : xs))
```

- Exercițiu!

# Rezumat

- **TDA**: Manieră de specificare a unui tip de date, alături de operațiile aferente și proprietățile acestora, independent de implementările concrete

- **Inducție structurală**: Instrument matematic pentru demonstrarea de noi proprietăți ale TDA-urilor, pornind de la cele cunoscute

- Uneori, necesitatea demonstrării de proprietăți ajutătoare