

# Paradigme de Programare

Conf. dr. ing. Andrei Olaru

andrei.olaru@upb.ro  
Departamentul de Calculatoare

2026

0

0 : 1

## Cursul 1: Introducere

- 1 Exemplu
- 2 Ce studiem la PP?
- 3 De ce studiem această materie?
- 4 Organizare
- 5 Paradigma de programare
- 6 Istoric: Paradigme și limbaje de programare
- 7 Introducere în Racket

Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 1

## Exemplu

APP

Exemplu Să se determine dacă un element  $e$  se regăsește într-o listă  $L$  ( $e \in L$ ).

Să se sorteze o listă  $L$ .

Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 3

Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 4

## Modelare funcțională (2)

APP

### Haskell

```

1 memList x [] = False
2 memList x (e:t) = x == e || memList x t
3
4 ins x [] = [x]
5 ins x l@(h:t) = if x < h then x:l else h : ins x t

```

Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 6

## Modelare logică

APP

### Prolog:

```

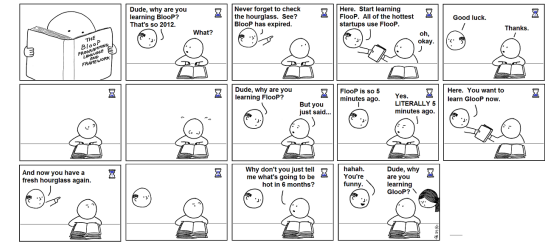
1 memberA(E, [_|_]) :- !.
2 memberA(E, [_|L]) :- memberA(E, L).
3
4 % elementul, lista, rezultatul
5 ins(E, [], [E]).
6 ins(E, [H | T], [E, H | T]) :- E < H, !.
7 ins(E, [H | T], [H | TE]) :- ins(E, T, TE).

```

Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 7

## Bloop and FlooP and Gloop

[[CC] BY-NC abstrusegoose.com] [http://abstrusegoose.com/503]



Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 2

## Modelare funcțională (1)

APP

### Racket:

```

1 (define memList (lambda (e L)
2   (if (null? L)
3       #f
4       (if (equal? (first L) e)
5           #t
6           (memList e (rest L))
7         )))
8
9
10 (define ins (lambda (x L)
11   (cond ((null? L) (list x))
12         ((< x (first L)) (cons x L))
13         (else (cons (first L) (ins x (rest L))))))

```

Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 5

## Ce studiem la PP?

Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 8

## Elemente pe care le vom studia

APP

- Paradigma funcțională și discuție despre paradigme de programare în general, în contrast cu paradigma imperativă și cea orientată-obiect.
- Tipuri de date abstracte
- Racket: introducere în [programare funcțională](#)
- [Calculul  \$\lambda\$](#)  ca bază teoretică a paradigmei funcționale
- Racket: [întârzierea](#) evaluării și fluxuri

- [Haskell](#): programare funcțională cu o sintaxă avansată
- Haskell: [evaluare leneșă și fluxuri](#)
- Haskell: [tipuri](#), sinteză de tip, și clase
- [Equational Reasoning](#)

Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 9

## De ce?

APP

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

The law of instrument – Abraham Maslow

Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 12

## De ce?

O bună cunoaștere a paradigmelor alternative → \$\$\$

APP

- [Developer Survey 2025](https://survey.stackoverflow.co/2025)
- [Developer Survey 2024](https://survey.stackoverflow.co/2024)

Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 15

## De ce studiem această materie?

Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 10

## De ce?

Mai concret

APP

· până acum ați studiat paradigma imperativă (legată și cu paradigma orientată-obiect)

→ **un anumit mod** de a privi procesul de rezolvare al unei probleme și de a căuta soluții la probleme de programare.

· paradigmele declarative studiate oferă o gamă diferită (complementară!) de **unelte** → **alte moduri** de a rezolva anumite probleme.

⇒ o pregătire ce permite accesul la poziții de calificare mai înaltă (arhitect, designer, etc.)

Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 13

## De ce?

Cine câștigă cel mai bine?

APP



Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 16

## De ce?

Ne vor folosi aceste lucruri în viața reală?

APP



The first math class.

The first math class.

[(C) Zach Weinersmith, Saturday Morning Breakfast Cereal]

[<https://www.smbc-comics.com/comic/a-new-method>]

Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 11

## De ce?

Sunt aceste paradigme relevante?

APP

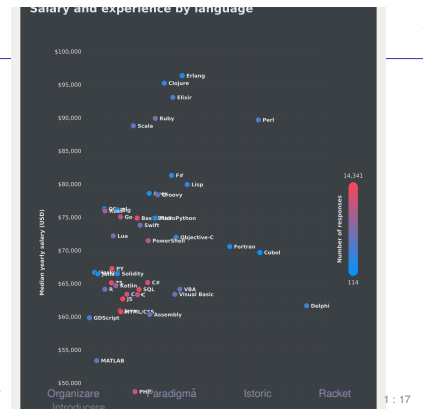
- **evaluarea leneșă** → prezentă în Python (de la v3), .NET (de la v4)
- **funcții anonime** → prezente în C++ (de la v11), C#/ .NET (de la v3.0/v3.5), Dart, Go, Java (de la JDK8), JS/ES, Perl (de la v5), PHP (de la v5.0.1), Python, Ruby, Swift.
- **Prolog și programarea logică** sunt folosite în software-ul modern de A.I., e.g. [Watson](#); automated theorem proving.
- În **industrie** sunt utilizate limbaje puternic funcționale precum [Erlang](#), [Scala](#), [F#](#), [Clojure](#).
- Limbaje **multi-paradigmă** → adaptarea paradigmei utilizate la necesități.

Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 14

## De ce?

Cine câștigă cel mai bine?

APP



## Organizare

Exemplu Ce? De ce? **Organizare** Introducere Paradigmă Istoric Racket 1 : 18

## Paradigma de programare

Exemplu Ce? De ce? Organizare Introducere **Paradigmă** Istoric Racket 1 : 21

## Ce vom studia? APP

Conținutul cursului

- 1 Diverse perspective conceptuale asupra noțiunii de calculabilitate efectivă → **modele de calculabilitate**.
- 2 Influența perspectivei alese asupra procesului de modelare și rezolvare a problemelor → **paradigme de programare**.
- 3 **Limbaje de programare** aferente paradigmelor, cu accent pe aspectul comparativ.

Exemplu Ce? De ce? Organizare Introducere **Paradigmă** Istoric Racket 1 : 24

## Unde găsec informații? APP

Resurse de bază

<https://ocw.cs.pub.ro/courses/pp>

Regulament: <https://ocw.cs.pub.ro/courses/pp/--/regulament>

Forumuri: Moodle → 03-ACS-L-CTI-Calculatoare-A2-S2-PP-CA-CB-CC  
<https://curs.upb.ro/2025/course/view.php?id=10195>

**Elementele cursului sunt comune la seriile CA, CB și CC.**

Exemplu Ce? De ce? **Organizare** Introducere Paradigmă Istoric Racket 1 : 19

## Ce înseamnă paradigma de programare APP

Ce diferă între paradigme?

- aceasta este o diferență între limbaje, dar este influențată și de natura paradigmei
- diferă sintaxa ← • mecanisme specifice unei paradigme aduc elemente noi de sintaxă  
e.g. funcțiile anonime
- diferă modul de construcție ← ce poate reprezenta o expresie, ce operatori putem aplica între expresii.
- diferă structura programului ← • ce anume reprezintă programul  
• cum se desfășoară execuția programului

Exemplu Ce? De ce? Organizare Introducere **Paradigmă** Istoric Racket 1 : 22

## Modele → paradigme → limbaje APP

Modele de calculabilitate

|                                     |                         |                    |
|-------------------------------------|-------------------------|--------------------|
| C, Pascal → procedural              | → paradigma imperativă  | → Mașina Turing    |
| Java, C++, Python → orientat-obiect |                         |                    |
| Racket, Haskell                     | → paradigma funcțională | → Mașina λ         |
| Prolog                              | → paradigma logică      | → FOL + Resolution |
| CLIPS                               | → paradigma asociativă  | → Mașina Markov    |

echivalente !

**T** | Teza Church-Turing: efectiv calculabil = Turing calculabil

Exemplu Ce? De ce? Organizare Introducere **Paradigmă** Istoric Racket 1 : 25

## Notare APP

mai multe la <https://ocw.cs.pub.ro/courses/pp/--/regulament>

- Laborator: 0.2p ← pentru străduință
- Teste grilă la laborator: 0.3p ← cu bonus până la 0.4p
- Teme (2): TBD ← cu bonusuri de până la 20%
- Activitate curs: 0.5p ← se punctează întrebările bune și foarte bune
- Test din materia de laborator: TBD  
*punctajele pe parcurs se trunchiază la 5p*
- Examen: 5p ← limbaje + teorie

Exemplu Ce? De ce? **Organizare** Introducere Paradigmă Istoric Racket 1 : 20

## Ce înseamnă paradigma de programare APP

Ce caracterizează o paradigmă?

- valorile de prim rang
- modul de construcție a programului
- modul de tipare al valorilor
- ordinea de evaluare (generare a valorilor)
- modul de legare al variabilelor (managementul valorilor)
- controlul execuției

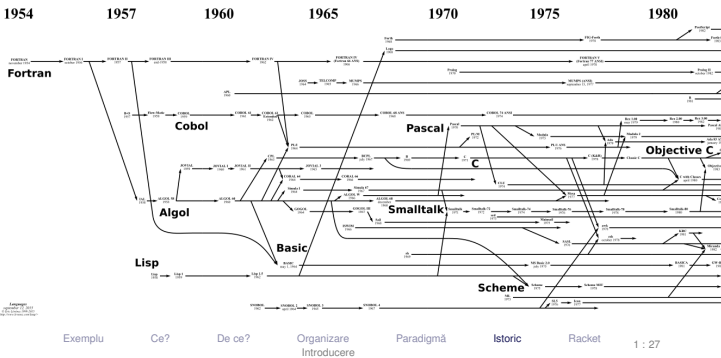
• **Paradigma de programare** este dată de stilul fundamental de construcție al structurii și elementelor unui program.

Exemplu Ce? De ce? Organizare Introducere **Paradigmă** Istoric Racket 1 : 23

## Istoric: Paradigme și limbaje de programare

## Istorie 1950-1975

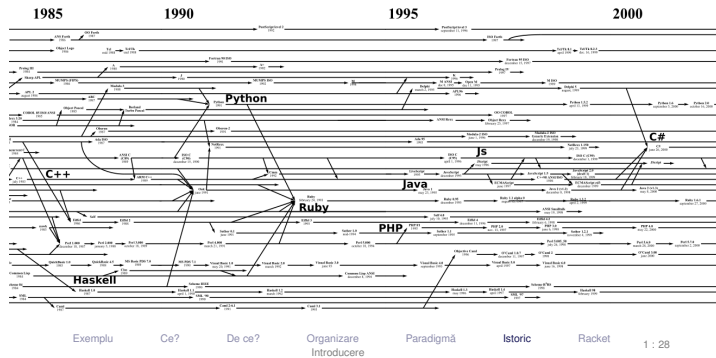
APP



Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 27

## Istorie 1975-1995

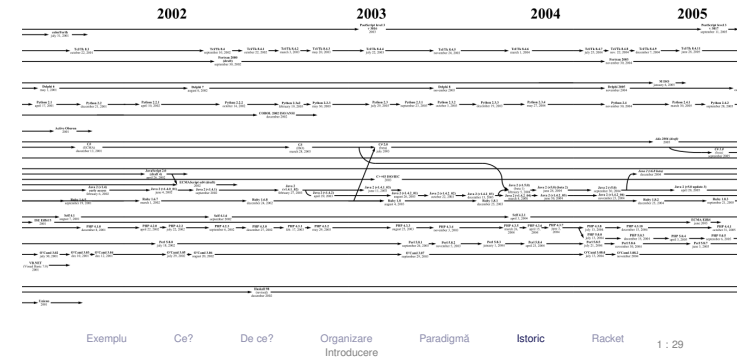
APP



Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 28

## Istorie 1995-2002

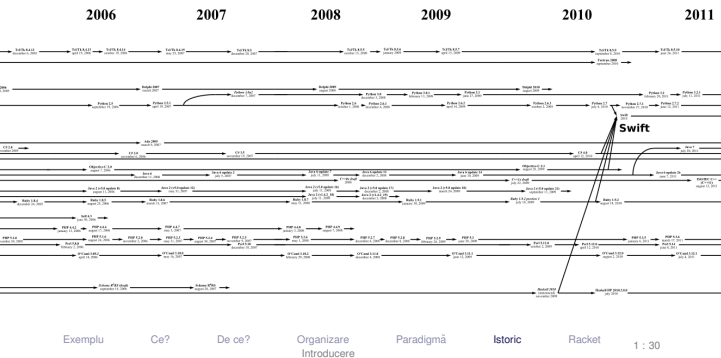
APP



Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 29

## Istorie 2002-2006

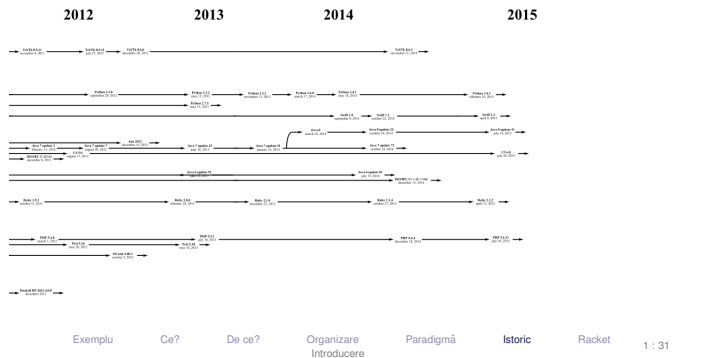
APP



Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 30

## Istorie 2006-2013

APP



Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 31

## Istorie Resurse

APP

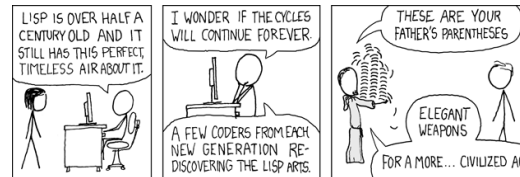
- imagine navigabilă (slides precedente): [<http://www.levenez.com/lang/>]
- Wikipedia:
  - [[http://en.wikipedia.org/wiki/Generational\\_list\\_of\\_programming\\_languages](http://en.wikipedia.org/wiki/Generational_list_of_programming_languages)]
  - [[https://en.wikipedia.org/wiki/Timeline\\_of\\_programming\\_languages](https://en.wikipedia.org/wiki/Timeline_of_programming_languages)]
  - [[https://en.wikipedia.org/wiki/Programming\\_paradigm#/media/File:Programming\\_paradigms.svg](https://en.wikipedia.org/wiki/Programming_paradigm#/media/File:Programming_paradigms.svg)]

Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 32

## Lisp cycles

[<http://xkcd.com/297/>]

APP



[[CC BY-NC Randall Munroe, xkcd.com]]

## Racket din 1975 (ca Scheme)

APP

- funcțional
- dialect de Lisp
- totul este văzut ca o funcție
- constante – expresii neevaluate
- perechi / liste pentru structurarea datelor
- apeluri de funcții – liste de apelare, evaluate
- evaluare aplicativă, funcții stricte, cu anumite excepții

Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 35

## Introducere în Racket

Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 33

Exemplu Ce? De ce? Organizare Introducere Paradigmă Istoric Racket 1 : 34

8 TDA

9 Constructori & Axiome

10 Inducție structurală

Structura unui TDA

Ce avem nevoie pentru a defini un TDA?

Operatii disponibile pe tipul de date → **Operații**

Exemplu

- numere naturale: adunare, scădere
- liste: adăugare, calcul lungime
- expresii aritmetice: evaluare, simplificare

Moduri de a construi valorile tipului de date → **Constructori**

Exemplu

- numere naturale: adunare, scădere, incrementare, verificare pentru 0
- liste: adăugare, eliminare, concatenare
- expresii aritmetice: aplicare operatori

**Axiome** care garanteaza funcționarea curentă a operațiilor

Constructorii de bază

Setul minim de constructori

+ **Constructor de bază** – parte a unui set de constructori care este **necesar și suficient** pentru a obține **toate valorile** din mulțimea de valori a tipului de date abstract.

- tipul Natural: **zero** : Natural ; **sucesor** : Natural → Natural
- tipul Listă: **null** : List ; **cons** : T × List → List
- tipul Expresie: **valoare** : Natural → Expresie ; **operatie** : Expresie × Operand × Expresie → Expresie

Exemplu

TDA

Tip de date abstract

Definiție

+ **Tip de date abstract – TDA** – Model matematic al unei **mulțimi** de valori și al **operațiilor** valide pe acestea.

Componente

- constructori de bază**: cum se generează valorile;
- operatori**: ce se poate face cu acestea;
- axiome**: cum lucrează operatorii / ce restricții există.

Constructorii de bază

Setul de constructori nu este unic

Pot avea și alte seturi de **constructori de bază**.

E.g. pentru listă pot avea

- null** : List
- singleton** : T → List
- concatenare** : List × List → List

Tipuri de date abstracte

Putem avea tipuri de date făcând abstracție de implementare?

- putem discuta despre **tipuri de date** fără a face apel la **implementarea** tipului și la modalitatea de **stocare** a datelor în memorie / pe disc
- definim tipurile de date într-un mod **matematic**
- avem nevoie să știm:
  - cum construim** valorile tipului de date?
  - cum operăm** cu valorile tipului de date?
- Ce este un număr natural?
- Ce este o listă?
- Ce este un arbore?
- Ce este o expresie aritmetică?

Exemplu

Constructorii & Axiome

Constructorii de bază

Tipuri

+ **Constructorii** pentru un tip T pot fi de tip:

- Nular** – 0 parametri  
Ex: zero, null
- Extern** – doar cu parametri care nu sunt de tip T  
Ex: singleton, valoare
- Intern** – cu cel puțin un parametru de tip T  
Ex: sucesor, cons, operatie

## Constrctori de bază

Dimensiunea valorilor

APP

+ **Dimensiunea** unei **valori** este numărul de **constrctori de bază** din formula care produce valoarea

- "2" :  $\text{succ}(\text{succ}(\text{zero}))$  — dimensiunea 3
- "[1, 7, 2]" :  $\text{cons}(1, \text{cons}(7, \text{cons}(2, \text{null})))$  — dimensiunea 4
- Constructorii **nulari** și **externi** produc întotdeauna valori de **dimensiune 1**
- Constructorii **interni** produc întotdeauna valori de **dimensiune > 1**

TDA      Constrctori & Axiome      Inducție structurală      2 : 10  
Tipuri de date abstracte

## Inducție structurală

TDA      Constrctori & Axiome      Inducție structurală      2 : 13  
Tipuri de date abstracte

## Inducție structurală

Exemplu (1)

APP

Pentru  $\text{add} : \text{Natural} \times \text{Natural} \rightarrow \text{Natural}$

cu axiomele

- ADD1  $\text{add}(\text{zero}, n) = n$
- ADD2  $\text{add}(\text{succesor}(m), n) = \text{succesor}(\text{add}(m, n))$

demonstrăm că

$\forall a, b, c \in \text{Natural} . \text{add}(a, \text{add}(b, c)) = \text{add}(\text{add}(a, b), c)$  (asociativitate)

**Caz de bază:**

$\text{add}(\text{zero}, \text{add}(b, c)) = \text{add}(b, c)$  (conform ADD1)

$\text{add}(\text{add}(\text{zero}, b), c) = \text{add}(b, c)$  (conform ADD1)      □

TDA      Constrctori & Axiome      Inducție structurală      2 : 16  
Tipuri de date abstracte

## Axiome

Exemple

APP

- $\text{concatenare}(\text{cons}(x, L1), L2) = \text{cons}(x, \text{concatenare}(L1, L2))$
- $\text{concatenare}(\text{null}, L) = L$
- $\text{adunare}(\text{succesor}(n), m) = \text{succesor}(\text{adunare}(n, m))$
- $\text{adunare}(\text{zero}, m) = m$
- $\text{evaluare}(\text{operatie}(E1, +, E2)) = \text{adunare}(\text{evaluare}(E1), \text{evaluare}(E2))$

Exemplu

TDA      Constrctori & Axiome      Inducție structurală      2 : 11  
Tipuri de date abstracte

## Inducție

Inducție matematică

APP

La **inducția matematică**, demonstrăm cazul de bază (e.g.  $n = 0$ ) și presupunând ca adevărată afirmație pentru valorile până la  $n - 1$ , demonstrăm adevărul afirmației pentru  $n$ .

Putem folosi inducția matematică pentru tipul *Natural*, dar pentru celelalte?

Pentru celelalte, folosim **inducția structurală**.

TDA      Constrctori & Axiome      Inducție structurală      2 : 14  
Tipuri de date abstracte

## Inducție structurală

Exemplu (2)

APP

cu axiomele

- ADD1  $\text{add}(\text{zero}, n) = n$
- ADD2  $\text{add}(\text{succesor}(m), n) = \text{succesor}(\text{add}(m, n))$

**Ipoteză inductivă:**  $\text{add}(x, \text{add}(b, c)) = \text{add}(\text{add}(x, b), c)$

**Pas de inducție:**

stânga:  $\text{add}(\text{succ}(x), \text{add}(b, c)) = \text{succ}(\text{add}(x, \text{add}(b, c)))$  (conform ADD2)

dreapta:  $\text{add}(\text{add}(\text{succ}(x), b), c) = \text{add}(\text{succ}(\text{add}(x, b)), c)$  (conform ADD2)  
 $= \text{succ}(\text{add}(\text{add}(x, b), c))$  (conform ADD2)

dar:  $\text{add}(x, \text{add}(b, c)) = \text{add}(\text{add}(x, b), c)$  (conform ipoteză)      □

TDA      Constrctori & Axiome      Inducție structurală      2 : 17  
Tipuri de date abstracte

## Axiome

Definiție

APP

+ **Axiomele** definesc comportamentul operatorilor pe toate valorile tipului.

Pentru fiecare operator avem nevoie de câte o **axiomă** pentru **fiecare constructor de bază**.

TDA      Constrctori & Axiome      Inducție structurală      2 : 12  
Tipuri de date abstracte

## Inducție structurală

Definiție

APP

+ **Inducția structurală** permite demonstrarea că dacă o afirmație despre o **operație** este adevărată pentru **cazurile de bază** (constructori nulari și interni) și dacă, presupunând că afirmația este adevărată pentru valorile de **dimensiune  $n - 1$** , putem demonstra că afirmația este adevărată pentru valorile de dimensiune  $n$ , atunci afirmația este adevărată **întotdeauna**.

TDA      Constrctori & Axiome      Inducție structurală      2 : 15  
Tipuri de date abstracte

## Cursul 3: Calcul Lambda

λ

- 11 Introducere
- 12 Lambda-expresii
- 13 Reducere
- 14 Evaluare
- 15 Limbajul lambda-0 și incursiune în TDA
- 16 Racket vs. lambda-0

Introducere      λ-Expresii      Reducere      Evaluare      λ<sub>0</sub> și TDA      Racket vs. λ<sub>0</sub>      3 : 1  
Calcul Lambda

# Introducere

# Modele de calculabilitate

De ce?

- ne punem problema dacă putem realiza un calcul sau nu → pentru a demonstra trebuie să avem un model simplu al calculului (**cum realizăm calculul**, în mod formal).
- un model de calculabilitate trebuie să fie cât mai simplu, atât ca număr de **operații** disponibile cât și ca mod de **construcție a valorilor**.
- corectitudinea unui program se demonstrează mai ușor dacă limbajul de programare este mai apropiat de mașina teoretică (modelul abstract de calculabilitate).

# Calculul Lambda

- **Model de calculabilitate** (Alonzo Church, 1932) – introdus în cadrul cercetărilor asupra fundamentelor matematicii. [http://en.wikipedia.org/wiki/Lambda\_calculus]
  - sistem formal pentru exprimarea calculului.
- **Echivalent** cu Mașina Turing (v. Teza Church-Turing)
- Axat pe conceptul matematic de **funcție** – totul este o funcție

# Aplicații ale calculului λ

- Aplicații importante în
  - **programare**
  - demonstrarea formală a **corectitudinii** programelor, datorită modelului simplu de execuție
- Baza teoretică a numeroase **limbaje**: LISP, Scheme, Haskell, ML, F#, Clean, Clojure, Scala, Erlang etc.

# Lambda-expresii

# λ-expresii Exemple

- 1  $x \rightarrow$  variabila (**numele**)  $x$
- 2  $\lambda x.x \rightarrow$  funcția **identitate**
- 3  $\lambda x.\lambda y.x \rightarrow$  funcție **selector**
- 4  $(\lambda x.x y) \rightarrow$  **aplicația** funcției identitate asupra parametrului actual  $y$
- 5  $(\lambda x.(x x) \lambda x.x) \rightarrow ?$

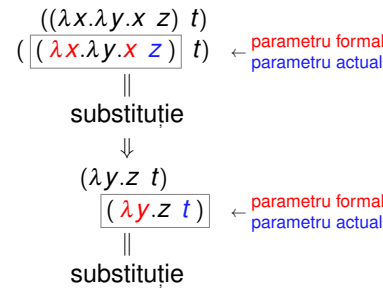
**Intuitiv**, evaluarea aplicației  $(\lambda x.x y)$  presupune **substituația textuală** a lui  $x$ , în corp, prin  $y \rightarrow$  rezultat  $y$ .

# λ-expresii Definiție

## + | λ-expresie

- **Variabilă**: o variabilă  $x$  este o λ-expresie;
- **Funcție**: dacă  $x$  este o variabilă și  $E$  este o λ-expresie, atunci  $\lambda x.E$  este o λ-expresie, reprezentând funcția **anonimă**, unară, cu parametrul formal  $x$  și corpul  $E$ ;
- **Aplicație**: dacă  $F$  și  $A$  sunt λ-expresii, atunci  $(F A)$  este o λ-expresie, reprezentând aplicația expresiei  $F$  asupra parametrului actual  $A$ .

# Evaluare Intuitiv



# Reducere

← nu mai este nicio funcție de aplicat

## $\beta$ -redex Cum arată (Formal, vedem mai târziu)

$\lambda$

- $\beta$ -redex: o  $\lambda$ -expresie de forma:  $(\lambda x.E A)$ 
  - $E$  –  $\lambda$ -expresie – este corpul funcției
  - $A$  –  $\lambda$ -expresie – este parametrul actual
- $\beta$ -redexul se reduce la  $E_{[A/x]}$  –  $E$  cu toate aparițiile **libere** ale lui  $x$  din  $E$  înlocuite cu  $A$  prin substituție textuală.

## Variabile Legate vs libere

$\lambda$

+ **O variabilă este legată** într-o expresie dacă **toate** aparițiile sale sunt legate în acea expresie.

+ **O variabilă este liberă** într-o expresie dacă nu este legată în acea expresie i.e. dacă **cel puțin o** apariție a sa este liberă în acea expresie.

- Atenție! În raport cu o **expresie dată!**

## Determinarea variabilelor libere și legate O abordare formală

$\lambda$

### Variabile libere (free variables)

- $FV(x) = \{x\}$
- $FV(\lambda x.E) = FV(E) \setminus \{x\}$
- $FV((E_1 E_2)) = FV(E_1) \cup FV(E_2)$

### Variabile legate (bound variables)

- $BV(x) = \emptyset$
- $BV(\lambda x.E) = BV(E) \cup \{x\}$
- $BV((E_1 E_2)) = BV(E_1) \setminus FV(E_2) \cup BV(E_2) \setminus FV(E_1)$

## Apariții ale variabilelor Legate vs libere

$\lambda$

+ **Apariție legată** O **apariție**  $x_n$  a unei variabile  $x$  este legată într-o expresie  $E$  dacă:

- $E = \lambda x.F$  sau
- $E = \dots \lambda x_n.F \dots$  sau
- $E = \dots \lambda x.F \dots$  și  $x_n$  apare în  $F$ .

+ **Apariție liberă** O **apariție** a unei variabile este liberă într-o expresie dacă nu este legată în acea expresie.

- Atenție! În raport cu o **expresie dată!**

## Variabile și Apariții ale lor Exemplu 1

$\lambda$

În expresia  $E = (\lambda x.x x)$ , evidențiem aparițiile lui  $x$ :

$(\lambda \underset{\langle 1 \rangle}{x} . \underset{\langle 2 \rangle}{x} \underset{\langle 3 \rangle}{x})$ .

- $x_{\langle 1 \rangle}, x_{\langle 2 \rangle}$  **legate** în  $E$
- $x_{\langle 3 \rangle}$  **liberă** în  $E$
- $x_{\langle 2 \rangle}$  **liberă** în  $F!$
- $x_{\langle 3 \rangle}$  **liberă** în  $E$  și  $F$

Exemplu

## Expresii închise

$\lambda$

+ **O expresie închisă** este o expresie care **nu** conține variabile libere.

Exemplu

- $(\lambda x.x \lambda x.\lambda y.x) \dots \rightarrow$  închisă
- $(\lambda x.x a) \dots \rightarrow$  deschisă, deoarece  $a$  este liberă

- Variabilele **libere** dintr-o  $\lambda$ -expresie pot sta pentru alte  $\lambda$ -expresii
- Înainte de evaluării, o expresie trebuie adusă la forma **închisă**.
- Procesul de înlocuire trebuie să se **termine**.

## Apariții ale variabilelor

$\lambda$

Mod de gândire

• O apariție **legată** în expresie este o apariție a parametrului formal al unei funcții definite **în** expresie, în corpul funcției; o apariție **liberă** este o apariție a parametrului formal al unei funcții definite **în exteriorul** expresiei, sau nu este parametru formal al niciunei funcții.

- $x_{\langle 1 \rangle}$  ← apariție liberă
- $(\lambda y. x_{\langle 1 \rangle} z)$  ← apariție încă liberă, nu o leagă nimeni
- $\lambda x_{\langle 2 \rangle} . (\lambda y. x_{\langle 1 \rangle} z)$  ←  $\lambda x_{\langle 2 \rangle}$  leagă apariția  $x_{\langle 1 \rangle}$
- $(\lambda x_{\langle 2 \rangle} . (\lambda y. x_{\langle 1 \rangle} z) x_{\langle 3 \rangle})$  ← apariția  $x_{\langle 3 \rangle}$  este liberă – este în exteriorul corpului funcției cu parametrul formal  $x$  ( $\lambda x_{\langle 2 \rangle}$ )
- $\lambda x_{\langle 4 \rangle} . (\lambda x_{\langle 2 \rangle} . (\lambda y. x_{\langle 1 \rangle} z) x_{\langle 3 \rangle})$  ←  $\lambda x_{\langle 4 \rangle}$  leagă apariția  $x_{\langle 3 \rangle}$

## Variabile și apariții ale lor Exemplu 2

$\lambda$

În expresia  $E = (\lambda x.\lambda z.(z x) (z y))$ , evidențiem aparițiile:

- $x_{\langle 1 \rangle}, x_{\langle 2 \rangle}, z_{\langle 1 \rangle}, z_{\langle 2 \rangle}$  **legate** în  $E$
- $y_{\langle 1 \rangle}, z_{\langle 3 \rangle}$  **libere** în  $E$
- $z_{\langle 1 \rangle}, z_{\langle 2 \rangle}$  **legate** în  $F$
- $x_{\langle 2 \rangle}$  **liberă** în  $F$
- $x_{\langle 1 \rangle}$  **legată** în  $E$ , dar **liberă** în  $F$
- $y_{\langle 1 \rangle}$  **liberă** în  $E$
- $z_{\langle 2 \rangle}$  **liberă** în  $E$ , dar **legată** în  $F$

Exemplu

## $\beta$ -reducere Definiție

$\lambda$

+  **$\beta$ -reducere:** Evaluarea expresiei  $(\lambda x.E A)$ , cu  $E$  și  $A$   $\lambda$ -expresii, prin **substituția textuală** a tuturor aparițiilor **libere** ale parametrului **formal** al funcției,  $x$ , din corpul acesteia,  $E$ , cu parametrul **actual**,  $A$ :

$$(\lambda x.E A) \rightarrow_{\beta} E_{[A/x]}$$

+  **$\beta$ -redex** Expresia  $(\lambda x.E A)$ , cu  $E$  și  $A$   $\lambda$ -expresii – o expresie pe care se poate aplica  $\beta$ -reducerea.

## $\beta$ -reducere

### Exemple

$\lambda$

- $(\lambda x.x y) \rightarrow_{\beta} x_{[y/x]} \rightarrow y$
  - $(\lambda x.\lambda x.x y) \rightarrow_{\beta} \lambda x.x_{[y/x]} \rightarrow \lambda x.x$
  - $(\lambda x.\lambda y.x y) \rightarrow_{\beta} \lambda y.x_{[y/x]} \rightarrow \lambda y.y$  **Greșit!** Variabila **liberă**  $y$  devine **legată**, schimbându-și semnificația.  $\rightarrow \lambda y^{(a)}.y^{(b)}$
- Care este problema?

Introducere  $\lambda$ -Expresii **Reducere** Evaluare  $\lambda_0$  și TDA Racket vs.  $\lambda_0$  3 : 20

## $\alpha$ -conversie

### Exemple

$\lambda$

- $\lambda x.(x y) \rightarrow_{\alpha} \lambda z.(z y) \rightarrow$  Corect!
- $\lambda x.\lambda x.(x y) \rightarrow_{\alpha} \lambda y.\lambda x.(x y) \rightarrow$  **Greșit!**  $y$  este liberă în  $\lambda x.(x y)$
- $\lambda x.\lambda y.(y x) \rightarrow_{\alpha} \lambda y.\lambda y.(y y) \rightarrow$  **Greșit!** Apariția liberă a lui  $x$  din  $\lambda y.(y x)$  devine legată, după substituție, în  $\lambda y.(y y)$
- $\lambda x.\lambda y.(y y) \rightarrow_{\alpha} \lambda y.\lambda y.(y y) \rightarrow$  Corect!

Introducere  $\lambda$ -Expresii **Reducere** Evaluare  $\lambda_0$  și TDA Racket vs.  $\lambda_0$  3 : 23

## Evaluare

Introducere  $\lambda$ -Expresii **Reducere** **Evaluare**  $\lambda_0$  și TDA Racket vs.  $\lambda_0$  3 : 26

## $\beta$ -reducere

### Coliziuni

$\lambda$

- **Problemă:** în expresia  $(\lambda x.E A)$ :
  - dacă variabilele libere din  $A$  nu au nume comune cu variabilele legate din  $E$ :  $FV(A) \cap BV(E) = \emptyset$   
 $\rightarrow$  reducere întotdeauna **corectă**
  - dacă există variabilele libere din  $A$  care au nume comune cu variabilele legate din  $E$ :  $FV(A) \cap BV(E) \neq \emptyset$   
 $\rightarrow$  reducere **potențial greșită**
- **Soluție:** redenumirea variabilelor legate din  $E$ , ce coincid cu cele libere din  $A \rightarrow$   **$\alpha$ -conversie**.

### Exemplu

$(\lambda x.\lambda y.x y) \rightarrow_{\alpha} (\lambda x.\lambda z.x y) \rightarrow_{\beta} \lambda z.x_{[y/x]} \rightarrow \lambda z.y$

Introducere  $\lambda$ -Expresii **Reducere** Evaluare  $\lambda_0$  și TDA Racket vs.  $\lambda_0$  3 : 21

## Reducere

### Definiții

$\lambda$

+ | **Pas de reducere:** O secvență formată dintr-o  $\alpha$ -conversie și o  $\beta$ -reducere, astfel încât a doua se produce **fără coliziuni**:  
 $E_1 \rightarrow E_2 \equiv E_1 \rightarrow_{\alpha} E_3 \rightarrow_{\beta} E_2$ .

+ | **Secvență de reducere:** Succesiune de zero sau mai mulți pași de reducere:  
 $E_1 \rightarrow^* E_2$ .  
Reprezintă un element din închiderea reflexiv-transitivă a relației  $\rightarrow$ .

Introducere  $\lambda$ -Expresii **Reducere** Evaluare  $\lambda_0$  și TDA Racket vs.  $\lambda_0$  3 : 24

## Întrebări

$\lambda$

Pentru construcția unei mașini de calcul

• Dacă am vrea să construim o mașină de calcul care să aibă ca program o  $\lambda$ -expresie și să aibă ca operație de bază pasul de reducere, ne punem câteva întrebări:

- 1 Când se **termină** calculul? Se termină **întotdeauna**?
- 2 Dacă mai multe secvențe de reducere se termină, obținem **întotdeauna același** rezultat?
- 3 Comportamentul **depinde** de secvența de reducere?
- 4 Dacă rezultatul este unic, **cum** îl obținem?

Introducere  $\lambda$ -Expresii **Reducere** **Evaluare**  $\lambda_0$  și TDA Racket vs.  $\lambda_0$  3 : 27

## $\alpha$ -conversie

### Definiție

$\lambda$

+ |  **$\alpha$ -conversie:** Redenumirea sistematică a variabilelor **legate** dintr-o funcție:  $\lambda x.E \rightarrow_{\alpha} \lambda y.E_{[y/x]}$ . Se impun două condiții.

### Exemplu

- $\lambda x.y \rightarrow_{\alpha} \lambda y.y_{[y/x]} \rightarrow \lambda y.y \rightarrow$  **Greșit!**
- $\lambda x.\lambda y.x \rightarrow_{\alpha} \lambda y.\lambda y.x_{[y/x]} \rightarrow \lambda y.\lambda y.y \rightarrow$  **Greșit!**

### : Condiții

- $y$  **nu** este o variabilă liberă, existentă deja în  $E$
- orice apariție liberă în  $E$  **rămâne** liberă în  $E_{[y/x]}$

Introducere  $\lambda$ -Expresii **Reducere** Evaluare  $\lambda_0$  și TDA Racket vs.  $\lambda_0$  3 : 22

## Reducere

### Proprietăți

$\lambda$

### : Reducere

- $E_1 \rightarrow E_2 \implies E_1 \rightarrow^* E_2$  – un pas este o secvență
- $E \rightarrow^* E$  – zero pași formează o secvență
- $E_1 \rightarrow^* E_2 \wedge E_2 \rightarrow^* E_3 \implies E_1 \rightarrow^* E_3$  – tranzitivitate

### Exemplu

$((\lambda x.\lambda y.(y x) y) \lambda x.x) \rightarrow (\lambda z.(z y) \lambda x.x) \rightarrow (\lambda x.x y) \rightarrow y$   
 $\rightarrow$   
 $((\lambda x.\lambda y.(y x) y) \lambda x.x) \rightarrow^* y$

Introducere  $\lambda$ -Expresii **Reducere** Evaluare  $\lambda_0$  și TDA Racket vs.  $\lambda_0$  3 : 25

## Terminarea reducerii (reductibilitate)

### Exemplu și definiție

$\lambda$

### Exemplu

$\Omega = (\lambda x.(x x) \lambda x.(x x)) \rightarrow (\lambda x.(x x) \lambda x.(x x)) \rightarrow^* \dots$

$\Omega$  **nu** admite nicio secvență de reducere care se termină.

+ | **Expresie reductibilă** este o expresie care admite (cel puțin o) secvență de reducere care se termină.

• expresia  $\Omega$  **nu** este reductibilă.

Introducere  $\lambda$ -Expresii **Reducere** **Evaluare**  $\lambda_0$  și TDA Racket vs.  $\lambda_0$  3 : 28

## Secvențe de reducere și terminare

λ

**Dar!**

$$E = (\lambda x. y \ \Omega)$$

$$\rightarrow y \quad \text{sau}$$

$$\rightarrow E \rightarrow y \quad \text{sau}$$

$$\rightarrow E \rightarrow E \rightarrow y \quad \text{sau} \dots$$

Exemplu

$$\vdots$$

$$\downarrow$$

$$y, n \geq 0$$

$$\vdots$$

- $E$  are o secvență de reducere care **nu** se termină;
- dar  $E$  are **forma normală**  $y \Rightarrow E$  este reducibilă;
- lungimea secvențelor de reducere ale  $E$  este **nemărginită**.

## Unicitatea formei normale Rezultate

λ

**T Teorema Church-Rosser / diamantului** Dacă  $E \rightarrow^* E_1$  și  $E \rightarrow^* E_2$ , atunci există  $E_3$  astfel încât  $E_1 \rightarrow^* E_3$  și  $E_2 \rightarrow^* E_3$ .



**C Corolar** Dacă o expresie este reducibilă, forma ei normală este **unică**. Ea corespunde **valorii** expresiei.

## Ce modalitate alegem?

λ

**T Teorema normalizării** Dacă o expresie este reducibilă, evaluarea **stânga-dreapta** a acesteia se termină.

- Teorema normalizării (normalizare = aducere la forma normală) **nu** garantează terminarea evaluării oricărei expresii, ci doar a celor **reducibile!**
- Dacă expresia este ireducibilă, **nicio** reducere nu se va termina.

## Forme normale

Cum știm că s-a terminat calculul?

λ

• Calculul **se termină** atunci când expresia nu mai poate fi redusă → expresia nu mai conține β-redecși.

+ **Forma normală** a unei expresii este o formă (la care se ajunge prin **reducere**, care **nu** mai conține β-redecși i.e. care **nu** mai poate fi redusă.

## Unicitatea formei normale

Exemplu

λ

Exemplu

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

- $\rightarrow \lambda z. ((\lambda x. x \ y) \ z) \rightarrow \lambda z. (y \ z) \rightarrow_{\alpha} \lambda a. (y \ a)$
- $\rightarrow (\lambda x. \lambda y. (x \ y) \ y) \rightarrow \lambda w. (y \ w) \rightarrow_{\alpha} \lambda a. (y \ a)$

- Forma normală corespunde unei **clase** de expresii, echivalente sub **redenumiri** sistematice.
  - **Valoarea** este un anumit membru al acestei clase de echivalență.
- ⇒ Valorile sunt **echivalente** în raport cu **redenumirea**.

## Răspunsuri la întrebări

λ

- 1 Când se **termină** calculul? Se termină **întotdeauna**?  
→ se termină cu **forma normală [funcțională]**. **NU** se termină decât dacă expresia este **reducibilă**.
- 2 Comportamentul **depinde** de secvența de reducere?  
→ **DA**.
- 3 Dacă mai multe secvențe de reducere se termină, obținem întotdeauna **același** rezultat?  
→ **DA**.
- 4 Dacă rezultatul este unic, **cum** îl obținem?  
→ Reducere **stânga-dreapta**.
- 5 Care este valoarea expresiei?  
→ Forma normală [funcțională] (**FN[F]**).

## Forme normale

Este necesar să mergem până la Forma Normală?

λ

+ **Forma normală funcțională – FNF** este o formă  $\lambda x. F$ , în care  $F$  poate conține β-redecși.

Exemplu

$$(\lambda x. \lambda y. (x \ y) \ (\lambda x. x)) \rightarrow_{FNF} \lambda y. (\lambda x. x \ y) \rightarrow_{FN} \lambda y. y$$

- FN a unei expresii închise este în mod necesar FNF.
- într-o FNF nu există o necesitate imediată de a **evalua** eventualii β-redecși interiori (funcția nu a fost încă aplicată).

## Modalități de reducere

Cum putem organiza reducerea?

λ

+ **Reducere stânga-dreapta**: Reducerea celui mai **superficial** și mai din **stânga** β-redec.

Exemplu

$$((\lambda x. x \ \lambda x. y) \ (\lambda x. (x \ x) \ \lambda x. (x \ x))) \rightarrow (\lambda x. y \ \Omega) \rightarrow y$$

+ **Reducere dreapta-stânga**: Reducerea celui mai **adânc** și mai din **dreapta** β-redec.

Exemplu

$$(\lambda x. (\lambda x. x \ \lambda x. y) \ (\lambda x. (x \ x) \ \lambda x. (x \ x))) \rightarrow (\lambda x. (\lambda x. x \ \lambda x. y) \ \Omega) \rightarrow \dots$$

## Ordine de evaluare

Tipuri

λ

- + **Evaluare aplicativă (eager)** – corespunde unei reduceri **mai degrabă dreapta-stânga**. Parametrii funcțiilor sunt evaluați **înaintea** aplicării funcției.
- + **Evaluare normală (lazy)** – corespunde reducerii **stânga-dreapta**. Parametrii funcțiilor sunt evaluați **la cerere**.
- + **Funcție strictă** – funcție cu evaluare **aplicativă**.
- + **Funcție nestrictă** – funcție cu evaluare **normală**.

- Evaluarea **aplicativă** prezintă în majoritatea limbajelor: C, Java, Scheme, PHP etc.

Exemplu

$(+ (+ 2 3) (* 2 3)) \rightarrow (+ 5 6) \rightarrow 11$

- Nevoie de funcții **nestrict**, chiar în limbajele aplicative: `if`, `and`, `or` etc.

Exemplu

$(\text{if } (< 2 3) (+ 2 3) (* 2 3)) \rightarrow (< 2 3) \rightarrow \#t \rightarrow (+ 2 3) \rightarrow 5$

Tipuri de date

Cum reprezentăm datele? Cum interpretăm valorile?

- Putem reprezenta toate datele prin funcții cărora, **convențional**, le dăm o semnificație **abstractă**.

Exemplu

$T \equiv_{\text{def}} \lambda x. \lambda y. x \quad F \equiv_{\text{def}} \lambda x. \lambda y. y$

- Pentru aceste **tipuri de date abstracte (TDA)** creăm operatori care transformă datele în mod coerent cu interpretarea pe care o dăm valorilor.

Exemplu

$\text{not} \equiv_{\text{def}} \lambda x. ((x F) T)$   
 $(\text{not } T) \rightarrow (\lambda x. ((x F) T) T) \rightarrow ((T F) T) \rightarrow F$

TDA Bool

Implementarea operatorilor

- $\text{if} \equiv_{\text{def}} \lambda c. \lambda x. \lambda y. ((c x) y)$
- $\text{and} \equiv_{\text{def}} \lambda x. \lambda y. ((x y) F)$ 
  - $((\text{and } T) a) \rightarrow ((\lambda x. \lambda y. ((x y) F) T) a) \rightarrow ((T a) F) \rightarrow a$
  - $((\text{and } F) a) \rightarrow ((\lambda x. \lambda y. ((x y) F) F) a) \rightarrow ((F a) F) \rightarrow F$
- $\text{or} \equiv_{\text{def}} \lambda x. \lambda y. ((x T) y)$ 
  - $((\text{or } T) a) \rightarrow ((\lambda x. \lambda y. ((x T) y) T) a) \rightarrow ((T T) a) \rightarrow T$
  - $((\text{or } F) a) \rightarrow ((\lambda x. \lambda y. ((x T) y) F) a) \rightarrow ((F T) a) \rightarrow a$
- $\text{not} \equiv_{\text{def}} \lambda x. ((x F) T)$ 
  - $(\text{not } T) \rightarrow (\lambda x. ((x F) T) T) \rightarrow ((T F) T) \rightarrow F$
  - $(\text{not } F) \rightarrow (\lambda x. ((x F) T) F) \rightarrow ((F F) T) \rightarrow T$

Limbajul lambda-0 și incursiune în TDA

TDA Bool  
Specificare

· Constructori:  $T : \rightarrow Bool$   
 $F : \rightarrow Bool$

· Operatori:  $\text{not} : Bool \rightarrow Bool$   
 $\text{and} : Bool^2 \rightarrow Bool$   
 $\text{or} : Bool^2 \rightarrow Bool$   
 $\text{if} : Bool \times A \times A \rightarrow A$

· Axiome:  $\text{not}(T) = F \quad \text{not}(F) = T$   
 $\text{and}(T, a) = a \quad \text{and}(F, a) = F$   
 $\text{or}(T, a) = T \quad \text{or}(F, a) = a$   
 $\text{if}(T, a, b) = a \quad \text{if}(F, a, b) = b$

TDA Pair

Implementare

- Intuiție: pereche → funcție ce așteaptă **selectorul**, pentru a-l aplica asupra membrilor
- $\text{fst} \equiv_{\text{def}} \lambda p. (p T)$ 
  - $(\text{fst } ((\text{pair } a) b)) \rightarrow (\lambda p. (p T) \lambda z. ((z a) b)) \rightarrow (\lambda z. ((z a) b) T) \rightarrow ((T a) b) \rightarrow a$
- $\text{snd} \equiv_{\text{def}} \lambda p. (p F)$ 
  - $(\text{snd } ((\text{pair } a) b)) \rightarrow (\lambda p. (p F) \lambda z. ((z a) b)) \rightarrow (\lambda z. ((z a) b) F) \rightarrow ((F a) b) \rightarrow b$
- $\text{pair} \equiv_{\text{def}} \lambda x. \lambda y. \lambda z. ((z x) y)$ 
  - $((\text{pair } a) b) \rightarrow ((\lambda x. \lambda y. \lambda z. ((z x) y) a) b) \rightarrow \lambda z. ((z a) b)$

- Am putea crea o mașină de calcul folosind calculul λ – mașină de calcul **ipotețică**;
- Mașina folosește limbajul λ<sub>0</sub> ≡ calcul lambda;
- Programul** → λ-expresie;
  - + Legări top-level de expresii la nume.
- Datele** → λ-expresii;
- Funcționarea mașinii → **reducere** – substituție textuală
  - evaluare normală;
  - terminarea evaluării cu forma normală funcțională;
  - se folosește numai expresii închise.

TDA Bool

Implementarea constructorilor de bază



**Intuiție:** bazat pe comportamentul necesar pentru `if`: **selecția** între cele două valori

- $T \equiv_{\text{def}} \lambda x. \lambda y. x$
- $F \equiv_{\text{def}} \lambda x. \lambda y. y$

TDA List și Natural

Implementare



**Intuiție:** listă → **pereche** (*head*, *tail*)

- $\text{nil} \equiv_{\text{def}} \lambda x. T$
- $\text{cons} \equiv_{\text{def}} \text{pair}$ 
  - $((\text{cons } e) L) \rightarrow ((\lambda x. \lambda y. \lambda z. ((z x) y) e) L) \rightarrow \lambda z. ((z e) L)$
- $\text{car} \equiv_{\text{def}} \text{fst} \quad \text{cdr} \equiv_{\text{def}} \text{snd}$



**Intuiție:** număr → **listă** cu lungimea egală cu valoarea numărului

- $\text{zero} \equiv_{\text{def}} \text{nil}$
- $\text{succ} \equiv_{\text{def}} \lambda n. ((\text{cons } \text{nil}) n)$
- $\text{pred} \equiv_{\text{def}} \text{cdr}$

· vezi și [http://en.wikipedia.org/wiki/Lambda\_calculus#Encoding\_datatypes]

## Absența tipurilor

Chiar avem nevoie de tipuri? – Rolul tipurilor

λ

- Modalitate de exprimare a **intenției** programatorului;
- **Documentare**: ce operatori acționează asupra căror obiecte;
- Reprezentarea **particulară** a valorilor de tipuri diferite: 1, "Hello", #t etc.;
- **Optimizarea** operațiilor specifice;
- Prevenirea **erorilor**;
- Facilitarea verificării **formale**;

Introducere λ-Expresii Reducere Calcul Lambda λ<sub>0</sub> și TDA Racket vs. λ<sub>0</sub> 3 : 47

## Absența tipurilor

Consecințe pozitive

λ

- **Flexibilitate** sporită în reprezentare;
- Potrivită în situațiile în care reprezentarea **uniformă** obiectelor, ca liste de simboluri, este convenabilă.

... vin cu prețul unei dificultăți sporite în **depanare, verificare și mentenanță**

Introducere λ-Expresii Reducere Calcul Lambda λ<sub>0</sub> și TDA Racket vs. λ<sub>0</sub> 3 : 50

## Combinator de punct fix

mai multe la [http://en.wikipedia.org/wiki/Lambda\\_calculus#Recursion\\_and\\_fixed\\_points](http://en.wikipedia.org/wiki/Lambda_calculus#Recursion_and_fixed_points)

λ

- Exemplu**
- $$Fix = \lambda f.(\lambda x.(f (x x)) \lambda x.(f (x x)))$$
- $(Fix F) \rightarrow (\lambda x.(F (x x)) \lambda x.(F (x x))) \rightarrow (F (\lambda x.(F (x x)) \lambda x.(F (x x)))) \rightarrow (F (Fix F))$
  - $(Fix F)$  este un **punct fix** al lui  $F$ .
  - $Fix$  se numește **combinator de punct fix**.

- $length \equiv_{def} (Fix Length) \sim (Length (Fix Length)) \sim \lambda L.(if (null? L) zero (succ ((Fix Length) (cdr L))))$

- Funcție recursivă, **fără** a fi textual recursivă!

Introducere λ-Expresii Reducere Calcul Lambda λ<sub>0</sub> și TDA Racket vs. λ<sub>0</sub> 3 : 53

## Absența tipurilor

Consecințe asupra reprezentării obiectelor

λ

- Un număr, o listă sau un arbore, posibil desemnate de **aceeași** valoare!
- Valori și operatori reprezentați de funcții, semnificația fiind dependentă de **context**.
- Valoare **aplicabilă** asupra unei alte valori → operator!

Introducere λ-Expresii Reducere Calcul Lambda λ<sub>0</sub> și TDA Racket vs. λ<sub>0</sub> 3 : 48

## Recursivitate

Perspective asupra recursivității

λ

• Cum realizăm recursivitatea în λ<sub>0</sub>, dacă nu avem nume de funcții?

- **Textuală**: funcție care se autoapelează, folosindu-și **numele**;
- **Semantică**: ce **obiect** matematic este desemnat de o funcție recursivă, cu posibilitatea construirii de funcții recursive **anonime**.

Introducere λ-Expresii Reducere Calcul Lambda λ<sub>0</sub> și TDA Racket vs. λ<sub>0</sub> 3 : 51

## Racket vs. lambda-0

## Absența tipurilor

Consecințe asupra corectitudinii calculului

λ

- Incapacitatea Mașinii λ de a
  - interpreta **semnificația** expresiilor;
  - asigura **corectitudinea** acestora (dpdv al tipurilor).
- Delegarea celor două aspecte **programatorului**;
- **Orice** operatori aplicabili asupra **oricărui** valori;
- Construcții eronate **acceptate** fără avertisment, dar calcule terminate cu
  - valori **fără** semnificație *sau*
  - expresii care **nu** sunt valori (nu au asociată o semnificație), dar sunt **irreductibile**
 → **instabilitate**.

Introducere λ-Expresii Reducere Calcul Lambda λ<sub>0</sub> și TDA Racket vs. λ<sub>0</sub> 3 : 49

## Implementare length

Problemă

λ

- Lungimea unei liste:
 
$$length \equiv_{def} \lambda L.(if (null? L) zero (succ (length (cdr L))))$$
- Cu ce **înlocuim** zona subliniată, pentru a evita recursivitatea textuală? (expresia pentru *length* nu este închisă!)
- Putem primi ca **parametru** o funcție echivalentă computațional cu *length*?
 
$$Length \equiv_{def} \lambda f L.(if (null? L) zero (succ (f (cdr L))))$$
- $(Length length) = length \rightarrow length$  este un **punct fix** al lui *Length*!
- Cum **obținem** punctul fix?

Introducere λ-Expresii Reducere Calcul Lambda λ<sub>0</sub> și TDA Racket vs. λ<sub>0</sub> 3 : 52

## Racket vs. λ<sub>0</sub>

Construcția expresiilor / sintaxă

λ

|                  | λ                  | Racket   |
|------------------|--------------------|--|
| Variabilă/nume   | $x$                | $x$  |
| Funcție          | $\lambda x.corp$   | $(lambda (x) corp)$                            |
| uncurry          | $\lambda x y.corp$ | $(lambda (x y) corp)$                          |
| Aplicare         | $(F A)$            | $(f a)$  |
| uncurry          | $(F A1 A2)$        | $(f a1 a2)$                                    |
| Legare top-level | -                  | $(define nume expr)$                           |
| Program          | λ-expresie închisă | colecție de legări top-level ( <i>define</i> ) |
| Valori           | λ-expresii / TDA   | valori de diverse tipuri (numere, liste, etc.) |

Introducere λ-Expresii Reducere Calcul Lambda λ<sub>0</sub> și TDA Racket vs. λ<sub>0</sub> 3 : 55

- similar cu  $\lambda_0$ , folosește S-expresii (bază Lisp);
- **tipat** – dinamic/latent
  - variabilele **nu** au tip;
  - valorile **au** tip (3, #f);
  - verificarea se face la **execuție**, în momentul aplicării unei funcții;
- evaluare **aplicativă**;
- permite recursivitate **textuală**;
- avem legări top-level.

## Introducere

# Variabile (Nume)

Proprietăți generale ale variabilelor

- Proprietăți
  - identificator
  - valoarea legată (la un anumit moment)
  - domeniul de vizibilitate (*scope*) + durata de viață
  - tip
- Stări
  - declarată: cunoaștem **identificatorul**
  - definită: cunoaștem și **valoarea** → variabila a fost *legată*

• În Racket, variabilele (numele) sunt legate *static* prin construcțiile `lambda`, `let`, `let*`, `letrec` și `define`, și sunt vizibile în domeniul construcției unde au fost definite (excepție face `define`).

# Cursul 4: Programare funcțională în Racket

- 17 Introducere
- 18 Legarea variabilelor
- 19 Evaluare
- 20 Construcția programelor prin recursivitate
- 21 Discuție despre tipare

## Analiza limbajului Racket

Ce analizăm la un limbaj de programare?

- Gestionarea valorilor
  - modul de tipare al valorilor
  - modul de legare al variabilelor
  - valorile de prim rang
- Gestionarea execuției
  - ordinea de evaluare (generare a valorilor)
  - controlul evaluării
  - modul de construcție al programelor

# Legarea variabilelor

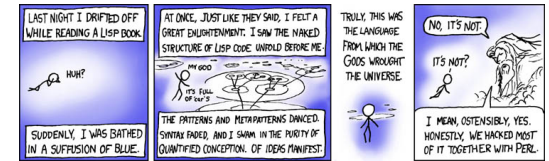
Definiții (1)

+ | **Legarea variabilelor** – modalitatea de **asociere** a apariției unei variabile cu definiția acesteia (deci cu valoarea).

+ | **Domeniul de vizibilitate** – *scope* – mulțimea punctelor din program unde o **definiție** (legare) este vizibilă.

# LISP

[[CC] BY-NC xkcd.com] [https://xkcd.com/224/]



## Legarea variabilelor

# Legarea variabilelor

Definiții (2)

+ | **Legare statică** – Valoarea pentru un nume este legată o singură dată, **la declarare**, în contextul în care aceasta a fost definită. Valoarea depinde doar de contextul **static** al variabilei.

- Domeniul de vizibilitate al legării poate fi desprins la **compilare**.

+ | **Legare dinamică** – Valorile variabilelor depind de **contextul de execuție** în care o expresie este **evaluată**.

## Legarea variabilelor în Racket



- Variabile definite în construcții interioare → **legate static, local**:
  - `lambda`
  - `let`
  - `let*`
  - `letrec`
- Variabile *top-level* → **legate static, global**:
  - `define`

Introducere    **Variabile**    Evaluare    Recursivitate    Tipare    4 : 9  
Programare funcțională în Racket

## Construcția `let`

Definiție, Exemplu, Semantică



- Leagă **static** variabile locale
- Sintaxă:

```
1 (let ((v1 e1) ... (vk ek) ... (vn en))
2   expr)
```
- Domeniul de vizibilitate a variabilei `vk` (cu valoarea `ek`): mulțimea punctelor din `expr` (**corp** `let`), în care aparițiile lui `vk` sunt **libere**.

Exemplu

```
1 (let ((x 1) (y 2)) (+ x 2))
```

• Atenție! Construcția `(let ((v1 e1) ... (vn en)) expr)` – **echivalentă** cu `(lambda (v1 ... vn) expr) e1 ... en`

Introducere    **Variabile**    Evaluare    Recursivitate    Tipare    4 : 12  
Programare funcțională în Racket

## Construcția `letrec`

Definiție



- Leagă **static** variabile locale
- Sintaxă:

```
1 (letrec ((v1 e1) ... (vk ek) ... (vn en))
2   expr)
```
- Domeniul de vizibilitate a variabilei `vk` = mulțimea punctelor din **întreaga** construcție, în care aparițiile lui `vk` sunt **libere**.

Introducere    **Variabile**    Evaluare    Recursivitate    Tipare    4 : 15  
Programare funcțională în Racket

## Construcția `lambda`

Definiție & Exemplu



- Leagă **static** parametrii formali ai unei funcții
- Sintaxă:

```
1 (lambda (p1 ... pk ... pn) expr)
```
- Domeniul de vizibilitate al parametrului `pk`: mulțimea punctelor din `expr` (care este **corpul funcției**), puncte în care apariția lui `pk` este **liberă**.

Introducere    **Variabile**    Evaluare    Recursivitate    Tipare    4 : 10  
Programare funcțională în Racket

## Construcția `let*`

Definiție & Exemplu



- Leagă **static** variabile locale
- Sintaxă:

```
1 (let* ((v1 e1) ... (vk ek) ... (vn en))
2   expr)
```
- Scope pentru variabila `vk` = mulțimea punctelor din
  - restul **legărilor** (legări ulterioare) și
  - **corp** – `expr`în care aparițiile lui `vk` sunt **libere**.

Exemplu

```
1 (let* ((x 1) (y x))
2   (+ x 2))
```

Introducere    **Variabile**    Evaluare    Recursivitate    Tipare    4 : 13  
Programare funcțională în Racket

## Construcția `letrec`

Exemplu



Exemplu

```
1 (letrec ((factorial
2         (lambda (n)
3           (if (zero? n) 1
4               (* n (factorial (- n 1)))))))
5   (factorial 5))
```

Introducere    **Variabile**    Evaluare    Recursivitate    Tipare    4 : 16  
Programare funcțională în Racket

## Construcția `lambda`

Semantică



- Aplicație:

```
1 ((lambda (p1 ... pn) expr)
2   a1 ... an)
```
- 1 Evaluare aplicativă: se evaluează **argumentele** `ak`, în ordine **aleatoare** (nu se garantează o anumită ordine).
- 2 Se evaluează **corpul** funcției, `expr`, ținând cont de legările `pk ← valoare(ak)`.
- 3 Valoarea aplicației este **valoarea** lui `expr`, evaluată mai sus.

Introducere    **Variabile**    Evaluare    Recursivitate    Tipare    4 : 11  
Programare funcțională în Racket

## Construcția `let*`

Semantică



```
1 (let* ((v1 e1) ... (vn en))
2   expr)
```

echivalent cu

```
1 (let ((v1 e1))
2   ...
3   (let ((vn en))
4     expr) ... )
```

- Evaluarea expresiilor `ei` se face **în ordine!**

Introducere    **Variabile**    Evaluare    Recursivitate    Tipare    4 : 14  
Programare funcțională în Racket

## Construcția `define`

Definiție & Exemplu



- Leagă **static** variabile **top-level**.
- Avantaje:
  - definirea variabilelor *top-level* în **orice** ordine
  - definirea de funcții **mutual** recursive

Definiții echivalente:

```
1 (define f1
2   (lambda (x)
3     (add1 x)
4   ))
5
6 (define (f2 x)
7   (add1 x)
8   ))
```

Introducere    **Variabile**    Evaluare    Recursivitate    Tipare    4 : 17  
Programare funcțională în Racket

## Evaluare

## Evaluarea în Racket

- Evaluare **aplicativă**: evaluarea parametrilor **înaintea** aplicării funcției asupra acestora (în ordine aleatoare).
- Funcții **stricte** (i.e.cu evaluare aplicativă)
  - Excepții: `if`, `cond`, `and`, `or`, `quote`.

## Controlul evaluării

- `quote` sau `'`
  - funcție **nestrictă**
  - întoarce parametrul **neevaluat**
- `eval`
  - funcție **strictă**
  - forțează **evaluarea** parametrului și întoarce valoarea acestuia

### Exemplu

```
1 (define sum '(+ 2 3))
2 sum ; '(+ 2 3)
3 (eval (list (car sum) (cadr sum) (caddr sum))) ; 5
```

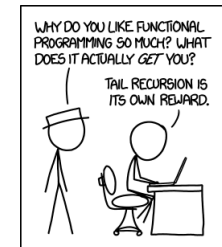
## Construcția programelor prin recursivitate

## Recursivitate

- **Recursivitatea** – element fundamental al paradigmei funcționale
  - Numai prin recursivitate (sau iterație) se pot realiza prelucrări pe date de dimensiuni nedefinite.
- Dar, este eficient să folosim recursivitatea?
  - recursivitatea (pe stivă) poate **încărca stiva**.

## Tail recursion

[[CC) BY-NC xkcd.com] [<https://xkcd.com/1270/>]



Alt text: Functional programming combines the flexibility and power of abstract mathematics with the intuitive clarity of abstract mathematics.

## Recursivitate

### Tipuri

- **pe stivă**:  $factorial(n) = n * factorial(n - 1)$ 
  - timp: liniar
  - spațiu: liniar (ocupat pe stivă)
  - dar, în procedural putem implementa factorialul în spațiu **constant**.
- **pe coadă**:  
 $factorial(n) = fH(n, 1)$   
 $fH(n, p) = fH(n - 1, p * n)$ ,  $n > 1$ ;  $p$  altfel
  - timp: liniar
  - spațiu: constant

- beneficiu **tail call optimization**

## Discuție despre tipare

## Tipuri în Racket

În Racket avem:

- **numere**: `1`, `2`, `1.5`
- **simboli (literali)**: `'abcd`, `'andrei`
- **valori booleene**: `#t`, `#f`
- **șiruri de caractere**: "șir de caractere"

- **perechi**: `(cons 1 2) → '(1 . 2)`
- **liste**: `(cons 1 (cons 2 '())) → '(1 2)`

- **funcții**: `(λ (e f) (cons e f)) → #<procedure>`

• Cum sunt gestionate tipurile valorilor (variabilelor) la **compilare** (verificare) și la **execuție**?

## Modalități de tipare



· Rolul tipurilor: exprimare a intenției programatorului, abstractizare, documentare, optimizare, verificare

+ **Tipare** – modul de gestionare a tipurilor.

· Clasificare după **momentul** verificării:

- statică
- dinamică

· Clasificare după **rigiditatea** regulilor:

- tare
- slabă

## Tipare tare vs. slabă

Exemple



· Clasificare după **libertatea** de a agrega valori de tipuri **diferite**.

### Exemplu Tipare tare

1 + "23" → **Eroare** (Haskell, Python)

### Exemplu Tipare slabă

1 + "23" = 24 (Visual Basic)  
1 + "23" = "123" (JavaScript)

## Întârzierea evaluării

## Tipare statică vs. dinamică

Exemplu



### Exemplu Tipare dinamică

JavaScript:  
var x = 5;  
if(condition) x = "here";  
print(x); → ce **tip** are x aici?

### Exemplu Tipare statică

Java:  
int x = 5;  
if(condition)  
x = "here"; → **Eroare la compilare**: x este int.  
print(x);

## Tiparea în Racket



· este **dinamică**

```
1 (if #t 'something (+ 1 #t)) → 'something  
2 (if #f 'something (+ 1 #t)) → Eroare
```

· este **tare**

```
1 (+ "1" 2) → Eroare
```

· dar, permite **liste** cu elemente de tipuri diferite.

## Motivație

De ce? → Luăm un exemplu



· Să se implementeze funcția **nestrictă** *prod*, astfel încât al doilea parametru să fie evaluat doar dacă primul este *true*:

- $prod(F, y) = 0$
- $prod(T, y) = y(y + 1)$

**Dar**, evaluarea parametrului *y* al funcției să se facă numai o singură dată.

· Problema de rezolvat: evaluarea **la cerere**.

## Tipare statică vs. dinamică

Caracteristici



### Tipare statică

- La compilare
- Valori și variabile
- Rulare mai rapidă
- Rigidă: sancționează orice construcție
- Debugging mai facil
- Declarații explicite sau inferențe de tip
- Pascal, C, C++, Java, Haskell

### Tipare dinamică

- La rulare
- Doar valori
- Rulare mai lentă (necesită verificarea tipurilor)
- Flexibilă: sancționează doar când este necesar
- Debugging mai dificil
- Permite metaprogramare (v. eval)
- Python, Scheme/Racket, Prolog, JavaScript, PHP

## Cursul 5: Evaluare leneșă în Racket



### 22 Întârzierea evaluării

### 23 Fluxuri

### 24 Căutare leneșă în spațiul stărilor

## Varianta 1

Încercare → implementare directă

```
1 (define prod  
2 (lambda (x y)  
3 (if x (* y (+ y 1)) 0)))  
4  
5 (define test  
6 (lambda (x)  
7 (let ((y 5))  
8 (prod x (and (display "y ") y))))))  
9 (test #f)  
10 (test #t)
```

Output: y 0 | y 30

- Implementarea nu respectă **specificația**, deoarece **ambii** parametri sunt evaluați în momentul aplicării

## Varianta 2

Încercare → quote & eval



```
1 (define prod
2   (lambda (x y)
3     (if x (* (eval y) (+ (eval y) 1)) 0)))
4
5 (define test
6   (lambda (x)
7     (let ((y 5))
8       (prod x (quote (and (display "y⊥") y))))))
9 (test #f)
10 (test #t)
```

Output: 0 | y undefined

- $x = \#f$  → comportament corect:  $y$  neevaluat
- $x = \#t$  → eroare: quote nu salvează contextul

Întârzierea evaluării      Fluxuri      Căutare în spațiul stărilor      5 : 5  
Evaluare leneșă în Racket

## Închideri funcționale

Definiție



+ **Închidere funcțională:** funcție care își salvează **contextul**, pe care îl va folosi, în momentul **aplicării**, pentru evaluarea corpului.

· **Notatie:** închiderea funcției  $f$  în contextul  $C \rightarrow \langle f; C \rangle$

Exemplu

$\langle \lambda x.z; \{z \leftarrow 2\} \rangle$

Întârzierea evaluării      Fluxuri      Căutare în spațiul stărilor      5 : 8  
Evaluare leneșă în Racket

## Promisiuni

Descriere



- Rezultatul încă **neevaluat** al unei expresii
- Valori de **prim rang** în limbaj
- **delay**
  - construiește o promisiune;
  - funcție nestrictă.
- **force**
  - forțează respectarea unei promisiuni, evaluând expresia doar la **prima** aplicare, și **salvându-i** valoarea;
  - începând cu a doua invocare, întoarce, direct, valoarea **memorată**.

Întârzierea evaluării      Fluxuri      Căutare în spațiul stărilor      5 : 11  
Evaluare leneșă în Racket

## Contexte computaționale

Definiție



+ **Context computațional** Contextul computațional al unui **punct**  $P$ , dintr-un program, la **momentul**  $t$ , este mulțimea variabilelor ale căror domenii de vizibilitate îl **conțin** pe  $P$ , la momentul  $t$ .

- Legare **statică** → mulțimea variabilelor care îl conțin pe  $P$  în domeniul **lexical** de vizibilitate
- Legare **dinamică** → mulțimea variabilelor definite cel mai recent, la **momentul**  $t$ , și referite din  $P$

Întârzierea evaluării      Fluxuri      Căutare în spațiul stărilor      5 : 6  
Evaluare leneșă în Racket

## Varianta 3

Încercare → închideri funcționale



```
1 (define prod
2   (lambda (x y)
3     (if x (* (y) (+ (y) 1)) 0))) ; (y)
4
5 (define test
6   (lambda (x)
7     (let ((y 5))
8       (prod x
9         (lambda () (and (display "y⊥") y))))))
10 (test #f)
11 (test #t)
```

Output: 0 | y y 30

- Comportament corect:  $y$  evaluat **la cerere** (deci leneș)
- $x = \#t$  →  $y$  evaluat de 2 ori → **ineficient**

Întârzierea evaluării      Fluxuri      Căutare în spațiul stărilor      5 : 9  
Evaluare leneșă în Racket

## Promisiuni

Proprietăți



- Salvarea **contextului computațional** al expresiei a cărei evaluare este întârziată și evaluarea ei ulterioară în **acel** context → asemănător cu închiderile funcționale.
- Salvarea **rezultatului** primei evaluări a expresiei.
- **Distingerea** primei forțări de celelalte → **efect lateral**, dar acceptabil din moment ce legările se fac static – nu pot exista valori care se schimbă *între timp*.

Întârzierea evaluării      Fluxuri      Căutare în spațiul stărilor      5 : 12  
Evaluare leneșă în Racket

## Contexte computaționale

Exemplu



Exemplu Ce variabile locale conține contextul computațional al punctului  $P$ ?

```
1 (lambda (x y)
2   (lambda (z)
3     (let ((x (car y)))
4       ; ...P...)))
```

Întârzierea evaluării      Fluxuri      Căutare în spațiul stărilor      5 : 7  
Evaluare leneșă în Racket

## Varianta 4

Promisiuni: delay & force



```
1 (define prod
2   (lambda (x y)
3     (if x (* (force y) (+ (force y) 1)) 0)))
4
5 (define test
6   (lambda (x)
7     (let ((y 5))
8       (prod x
9         (delay (and (display "y⊥") y))))))
10 (test #f)
11 (test #t)
```

Output: 0 | y 30

- Rezultat corect:  $y$  evaluat **la cerere**, o **singură dată** → **evaluare leneșă eficientă**

Întârzierea evaluării      Fluxuri      Căutare în spațiul stărilor      5 : 10  
Evaluare leneșă în Racket

## Evaluare întârziată

Abstractizare a implementării cu promisiuni



Exemplu Continuare a exemplului cu funcția prod

```
1 (define-syntax-rule (pack expr) (delay expr))
2
3 (define unpack force)
4
5 (define prod (lambda (x y)
6   (if x (* (unpack y) (+ (unpack y) 1)) 0)))
7 (define test (lambda (x)
8   (let ((y 5))
9     (prod x (pack (and (display "y⊥") y))))))
```

· utilizarea nu depinde de implementare (am definit funcțiile **pack** și **unpack** care **abstractizează** implementarea concretă a evaluării întârziate.

Întârzierea evaluării      Fluxuri      Căutare în spațiul stărilor      5 : 13  
Evaluare leneșă în Racket

## Evaluare întârziată

Abstractizare a implementării cu închideri

Continuare a exemplului cu funcția prod

```

1 (define-syntax-rule (pack expr) (lambda () expr) )
2
3 (define unpack (lambda (p) (p)))
4
5 (define prod (lambda (x y)
6   (if x (* (unpack y) (+ (unpack y) 1)) 0)))
7 (define test (lambda (x)
8   (let ((y 5))
9     (prod x (pack (and (display "y_") y)))) )))

```

utilizarea nu depinde de implementare (același cod ca și anterior, altă implementare a funcționalității de evaluare întârziată, acum mai puțin eficientă).

## Fluxuri

## Motivație

Observații

- Varianta 1 – iterativă (d.p.d.v. proces):
  - **eficientă**, datorită spațiului suplimentar constant;
  - **ne-elegantă** → trebuie să implementăm generarea numerelor.
- Varianta 2 – folosește liste:
  - **ineficientă**, datorită spațiului posibil mare, ocupat la un moment dat – toate numerele din interval [a, b].
  - **elegantă** și concisă;
- Cum **îmbinăm** avantajele celor 2 abordări? Putem stoca **procesul** fără a stoca **rezultatul** procesului?



## Fluxuri

Operatori: construcție și selecție

- cons, car, cdr, nil, null?

```

1 (define-syntax-rule (stream-cons head tail)
2   (cons head (pack tail)))
3
4 (define stream-car car)
5
6 (define stream-cdr (lambda (s)
7   (unpack (cdr s))))
8
9 (define stream-nil '())
10
11 (define stream-null? null?)

```

## Fluxuri

Caracteristici

- Secvențe construite **parțial**, extinse la cerere, ce creează **iluzia** completitudinii structurii;
- Îmbinarea **eleganței** manipulării listelor cu **eficiența** calculului incremental;
- Bariera de abstractizare:
  - componentele **listelor** evaluate la **construcție** (cons)
  - componentele **fluxurilor** evaluate la **selecție** (cdr)
- Construcție și utilizare:
  - **separate** la nivel conceptual → **modularitate**;
  - **întrepătrunse** la nivel de proces (utilizarea necesită construcția concretă).

## Fluxuri – Exemple

Implementarea unui flux de numere 1

- Definiție cu închideri:
 

```
(define ones (lambda () (cons 1 (lambda () (ones)))))
```
- Definiție cu fluxuri:
 

```
1 (define ones (stream-cons 1 ones))
2 (stream-take 5 ones) ; (1 1 1 1 1)
```
- Definiție cu promisiuni:
 

```
(define ones (delay (cons 1 ones)))
```

## Motivație

Luăm un exemplu

Determinați suma numerelor pare<sup>1</sup> din intervalul [a, b].

```

1 (define even-sum-iter ; varianta 1
2   (lambda (a b)
3     (let iter ((n a)
4               (sum 0))
5       (cond ((> n b) sum)
6             ((even? n) (iter (+ n 1) (+ sum n)))
7             (else (iter (+ n 1) sum)))))
8
9 (define even-sum-lists ; varianta 2
10  (lambda (a b)
11    (foldl + 0 (filter even? (interval a b)))))

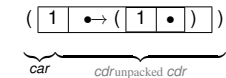
```

<sup>1</sup>stă pentru o verificare o potențial mai complexă, e.g. numere prime

## Fluxuri


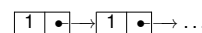
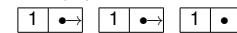
Intuitiv

- o listă este o **pereche**;
- explorarea listei se face prin operatorii **car** – primul element – și **cdr** – **restul** listei;
- am dori să **generăm** cdr algoritmic, dar **la cerere**.



## Fluxuri – Exemple

Flux de numere 1 – discuție

- Ca proces:
 
- Structural:
 
- Extinderea se realizează în spațiu constant:
 

## Fluxul numerelor naturale

### Formulare explicită

```
1 (define naturals-from (lambda (n)
2   (stream-cons n (naturals-from (+ n 1)))))
3
4 (define naturals (naturals-from 0))
```

```
1 (define naturals
2   (stream-cons 0
3     (stream-zip-with + ones naturals)))
```

#### Atenție:

- Închideri: multiple parcurgeri ale fluxului determină **reevaluarea** porțiunilor deja explorate.
- Promisiuni: parcurgerea fluxului determină evaluarea **dincolo** de porțiunile deja explorate.

Întârzierea evaluării      Fluxuri      Căutare în spațiul stărilor      5 : 23  
Evaluare leneșă în Racket

## Fluxul numerelor prime

### Implementare

```
1 (define sieve (lambda (s)
2   (if (stream-null? s) s
3       (stream-cons (stream-car s)
4                     (sieve (stream-filter
5                             (lambda (n) (not (zero?
6                                             (remainder n (stream-car s))))
7                             (stream-cdr s)
8                             )))
9   )))
10
11 (define primes (sieve (naturals-from 2)))
```

Întârzierea evaluării      Fluxuri      Căutare în spațiul stărilor      5 : 26  
Evaluare leneșă în Racket

## Specificarea unei probleme

### Aplicație pe $Pal_n$

- Starea **inițială**: șirul vid
- Operatorii de generare a stărilor **succesor** ale unei stări: inserarea unui caracter la începutul unui șir dat
- Operatorul de verificare a proprietății de **scop** a unei stări: palindrom

Întârzierea evaluării      Fluxuri      Căutare în spațiul stărilor      5 : 29  
Evaluare leneșă în Racket

## Fluxul numerelor pare

### În două variante

```
1 (define even-naturals
2   (stream-filter even? naturals))
3
4 (define even-naturals
5   (stream-zip-with + naturals naturals))
```

Întârzierea evaluării      Fluxuri      Căutare în spațiul stărilor      5 : 24  
Evaluare leneșă în Racket

## Căutare leneșă în spațiul stărilor

## Căutare în spațiul stărilor

- Spațiul stărilor ca **graf**:
  - noduri: **stări**
  - muchii (orientate): **transformări** ale stărilor în stări succesori
- Posibile strategii de **căutare**:
  - lățime: **completă** și optimală
  - adâncime: **incompletă** și suboptimală

Întârzierea evaluării      Fluxuri      Căutare în spațiul stărilor      5 : 30  
Evaluare leneșă în Racket

## Fluxul numerelor prime

### Metodă

- Ciurul lui **Eratostene**.
- Pornim de la fluxul numerelor **naturale**, începând cu 2.
- Elementul **curent** din fluxul inițial aparține fluxului numerelor prime.
- **Restul** fluxului generat se obține
  - eliminând **multipli** elementului curent din fluxul inițial;
  - continuând procesul de **filtrare**, cu elementul următor.

Întârzierea evaluării      Fluxuri      Căutare în spațiul stărilor      5 : 25  
Evaluare leneșă în Racket

## Spațiul stărilor unei probleme

+ **Spațiul stărilor unei probleme** Mulțimea configurațiilor valide din universul problemei.

**Exemplu** Fie problema  $Pal_n$ : *Să se determine palindroamele de lungime cel puțin  $n$ , ce se pot forma cu elementele unui alfabet fixat.*

**Stările** problemei → toate șirurile generabile cu elementele alfabetului respectiv.

## Căutare în lățime

### Obişnuită

```
1 (define breadth-search-goal
2   (lambda (init expand goal?)
3     (letrec ((search (lambda (states)
4                       (if (null? states) '()
7                           (let ((state (car states)) (states (cdr states)))
8                             (if (goal? state) state
9                                 (search (append states (expand state))))
9   )))))
10   (search (list init))))
```

- Generarea unei **singure** soluții
- Cum le obținem pe **celelalte**, mai ales dacă spațiul e **infini**?

Întârzierea evaluării      Fluxuri      Căutare în spațiul stărilor      5 : 31  
Evaluare leneșă în Racket

## Căutare în lățime

Leneșă (1) – fluxul stărilor *scop*



## Căutare în lățime

Leneșă (2)



## Cursul 6: Programare funcțională în Haskell



```
1 (define lazy-breadth-search (lambda (init expand)
2   (letrec ((search (lambda (states)
3     (if (stream-null? states) states
4       (let ((state (stream-car states))
5         (states (stream-cdr states)))
6         (stream-cons state
7           (search (stream-append states
8             (expand state))))
9       )))))
10 (search (stream-cons init stream-nil))
11 )))
```

Întărierea evaluării

Fluxuri  
Evaluare leneșă în Racket

Căutare în spațiul stărilor 5 : 32

```
1 (define lazy-breadth-search-goal
2   (lambda (init expand goal?)
3     (stream-filter goal?
4       (lazy-breadth-search init expand))
5   ))
```

- Nivel înalt, conceptual: **separare** între explorarea spațiului și identificarea stărilor *scop*.
- Nivel scăzut, al instrucțiunilor: **întrepătrunderea** celor două aspecte.
- Aplicații:
  - Palindroame
  - Problema regiunilor

Întărierea evaluării

Fluxuri  
Evaluare leneșă în Racket

Căutare în spațiul stărilor 5 : 33

25 Introducere

26 Sintaxă

27 Evaluare

Introducere

Sintaxă  
Programare funcțională în Haskell

Evaluare 6 : 1

## Haskell

[[CC] BY-NC xkcd.com] [https://xkcd.com/1312/]



Alt text: The problem with Haskell is that it's a language built on lazy evaluation and nobody's actually called for it.

Introducere

Sintaxă  
Programare funcțională în Haskell

Evaluare 6 : 2

## Introducere

Introducere

Sintaxă  
Programare funcțională în Haskell

Evaluare 6 : 3

## Haskell

[https://en.wikipedia.org/wiki/Haskell\_(programming\_language)]



- din 1990;
- GHC – Glasgow Haskell Compiler (The Glorious Glasgow Haskell Compilation System)
  - dialect Haskell standard *de facto*;
  - compilează în/folosind C;
- Haskell Stack
- nume dat după logicianul Haskell Curry;
- aplicații: Pugs, Darcs, Linspire, Xmonad, Cryptol, seL4, Pandoc, web frameworks.

Introducere

Sintaxă  
Programare funcțională în Haskell

Evaluare 6 : 4

## Paralelă între limbaje



| Criteriu                | Racket                          | Haskell                 |
|-------------------------|---------------------------------|-------------------------|
| Funcții                 | <i>Curry</i> sau <i>uncurry</i> | <i>Curry</i>            |
| Tipare                  | Dinamică, tare (-liste)         | <b>Statică, tare</b>    |
| Legarea variabilelor    | Statică                         | Statică                 |
| Evaluare                | Aplicativă                      | <b>Normală (Lenesă)</b> |
| Transferul parametrilor | <i>Call by sharing</i>          | <i>Call by need</i>     |
| Efecte laterale         | set!*                           | Interzise               |

Introducere

Sintaxă  
Programare funcțională în Haskell

Evaluare 6 : 5

## Sintaxă

Introducere

Sintaxă  
Programare funcțională în Haskell

Evaluare 6 : 6

## Funcții



- toate funcțiile sunt *Curry*;
- aplicabile asupra **oricâtor** parametri la un moment dat.

**Exemplu** : Definiții **echivalente** ale funcției *add*:

```
1 add1      = \x y -> x + y
2 add2      = \x -> \y -> x + y
3 add3 x y  = x + y
4
5 result    = add1 1 2  -- echivalent, ((add1 1) 2)
6 result2   = add3 1 2  -- echivalent, ((add3 1) 2)
7 inc       = add1 1
```

Introducere

Sintaxă  
Programare funcțională în Haskell

Evaluare 6 : 7

## Funcții vs operatori



- Aplicabilitatea **parțială** a operatorilor infixaiți
- **Transformări** operator → funcție și funcție → operator

**Exemplu** Definiții **echivalente** ale funcțiilor `add` și `inc`:

```
1 add4 = (+)
2 result1 = (+) 1 2
3 result2 = 1 'add4' 2
4
5 inc1 = (1 +)
6 inc2 = (+ 1)
7 inc3 = (1 'add4')
8 inc4 = ('add4' 1)
```

Introducere Sintaxă Programare funcțională în Haskell Evaluare 6 : 8

## Evaluare

Introducere Sintaxă Programare funcțională în Haskell Evaluare 6 : 11

## Pași în aplicarea funcțiilor

Ordine



- 1 **Pattern matching**: evaluarea parametrilor **suficient** cât să se constate (ne-)potrivirea cu *pattern*-ul;
- 2 Evaluarea **gărzilor** ( `|` );
- 3 Evaluarea variabilelor **locale**, **la cerere** (`where`, `let`).

Introducere Sintaxă Programare funcțională în Haskell Evaluare 6 : 14

## Pattern matching



- Definirea comportamentului funcțiilor pornind de la **structura** parametrilor → traducerea axiomelor TDA.

**Exemplu**

```
1 add5 0 y = y -- add5 1 2
2 add5 (x + 1) y = 1 + add5 x y
3
4 sumList [] = 0 -- sumList [1,2,3]
5 sumList (hd:t1) = hd + sumList t1
6
7 sumPair (x, y) = x + y -- sumPair (1,2)
8
9 sumTriplet (x, y, z@(hd:_)) = -- sumTriplet
10 x + y + hd + sumList z -- (1,2,[3,4,5])
```

Introducere Sintaxă Programare funcțională în Haskell Evaluare 6 : 9

## Evaluare



- Evaluare **leneșă**: parametri evaluați **la cerere**, **cel mult** o dată, eventual **parțial**, în cazul obiectelor structurate
- Transferul parametrilor: *call by need*
- Funcții **nestricte**!

**Exemplu**

```
1 f (x, y) z = x + x
```

Evaluare:

```
1 f (2 + 3, 3 + 5) (5 + 8)
2 → (2 + 3) + (2 + 3)
3 → 5 + 5 reutilizăm rezultatul primei evaluări!
4 → 10 ceilalți parametri nu sunt evaluați
```

Introducere Sintaxă Programare funcțională în Haskell Evaluare 6 : 12

## Pași în aplicarea funcțiilor



Exemplu – revisited

**Exemplu** execuția exemplului anterior

```
1 frontInterval 3 5          evaluare pattern
2 ?? notNil xs              evaluare prima gardă
3 ??     where              necesar xs → evaluare where
4 ??     xs = [3 .. 5]
5 ??     → 3:[4 .. 5]
6 ?? → notNil (3:[4 .. 5])
7 ?? → True
8 → frontSum xs              evaluare valoare gardă
9     where
10    xs = 3:[4 .. 5]         xs deja calculat
11    → 3:4:[5]
12 → frontSum (3:4:[5])
13 → 3 + 4 → 7
```

Introducere Sintaxă Programare funcțională în Haskell Evaluare 6 : 15

## List comprehensions



- Definirea listelor prin **proprietățile** elementelor, ca într-o specificare matematică

**Exemplu**

```
1 squares lst = [x * x | x <- lst]
2
3 quickSort [] = []
4 quickSort (h:t) = quickSort [x | x <- t, x <= h]
5 ++ [h]
6 ++ quickSort [x | x <- t, x > h]
7
8 interval = [0 .. 10]
9 evenInterval = [0, 2 .. 10]
10 naturals = [0 ..]
```

Introducere Sintaxă Programare funcțională în Haskell Evaluare 6 : 10

## Pași în aplicarea funcțiilor

Exemplu



**Exemplu**

```
1 frontSum (x:y:zs) = x + y
2 frontSum [x] = x
3
4 notNil [] = False
5 notNil (_:_) = True
6
7 frontInterval m n
8 | notNil xs = frontSum xs
9 | otherwise = n
10 where
11     xs = [m .. n]
```

Introducere Sintaxă Programare funcțională în Haskell Evaluare 6 : 13

## Consecințe



- Evaluarea **parțială** a structurilor – liste, tupluri etc.
- Listele sunt, **implicit**, văzute ca **fluxuri**!

**Exemplu**

```
1 ones = 1 : ones
2
3 naturalsFrom n = n : (naturalsFrom (n + 1))
4 naturals1 = naturalsFrom 0
5 naturals2 = 0 : (zipWith (+) ones naturals2)
6
7 evenNaturals1 = filter even naturals1
8 evenNaturals2 = zipWith (+) naturals1 naturals2
9
10 fibo = 0 : 1 : (zipWith (+) fibo (tail fibo))
```

Introducere Sintaxă Programare funcțională în Haskell Evaluare 6 : 16



28 Tipare

29 Sinteză de tip

30 TDA

Tipuri Exemple de valori



Exemplu

```
1 5 :: Integer
2 'a' :: Char
3 (+1) :: Integer -> Integer
4 [1,2,3] :: [Integer] -- liste de un singur tip !
5 (True, "Hello") :: (Bool, [Char])
6 etc.
```

- Tipurile de bază sunt tipurile elementare din limbaj: Bool, Char, Integer, Int, Float, ...

• Reprezentare uniformă:

```
1 data Integer = ... | -2 | -1 | 0 | 1 | 2 | ...
2 data Char = 'a' | 'b' | 'c' | ...
```

Sinteză de tip

Tipare

Construcții de tip

⇒ tipuri noi pentru valori sau funcții

- Funcții de tip, ce îmbogățesc tipurile din limbaj.

Exemplu Construcții de tip predefinite

```
1 -- Constructorul de tip functie: ->
2 (-> Bool Bool) => Bool -> Bool
3 (-> Bool (Bool -> Bool)) => Bool -> (Bool -> Bool)
4
5 -- Constructorul de tip lista: []
6 ([] Bool) => [Bool]
7 ([] [Bool]) => [[Bool]]
8
9 -- Constructorul de tip tuplu: (,...)
10 ((,) Bool Char) => (Bool, Char)
11 ((,,) Bool ((,) Char [Bool]) Bool)
12 => (Bool, (Char, [Bool]), Bool)
```

Sinteză de tip

Definiție

+ | **Sinteză de tip – type inference** – Determinarea automată a tipului unei expresii, pe baza unor reguli precise.

- Adnotările **explicite** de tip, deși posibile, **necesare** în majoritatea cazurilor
- Dependentă de:
  - **componentele** expresiei
  - **contextul** lexical al expresiei
- Reprezentarea tipurilor → **expresii** de tip:
  - **constante** de tip: tipuri de bază;
  - **variabile** de tip: pot fi legate la orice expresii de tip;
  - **aplicații** ale constructorilor de tip pe expresii de tip.

Tipuri

Pentru toate valorile (inclusiv funcții)



- Tipuri ca **mulțimi** de valori:

- Bool = {True, False}
- Natural = {0, 1, 2, ...}
- Char = {'a', 'b', 'c', ...}

- **Rolul** tipurilor (vezi cursuri anterioare);

- Tipare **statică**:

- etapa de tipare **anterioară** etapei de evaluare;
- asocierea **fiecărei** expresii din program cu un tip;

- Tipare **tare**: absența conversiilor **implicite** de tip;

- Expresii de:

- **program**: 5, 2 + 3, x && (not y)
- **tip**: Integer, [Char], Char -> Bool, a

Construcții de tip

Tipurile funcțiilor



- Constructorul -> este asociativ **dreapta**:

```
Integer -> Integer -> Integer
≡ Integer -> (Integer -> Integer)
```

Exemplu

```
1 add6 :: Integer -> Integer -> Integer
2 add6 x y = x + y
3
4 f :: (Integer -> Integer) -> Integer
5 f g = (g 3) + 1
6
7 idd :: a -> a -- functie polimorfica
8 idd x = x -- a: variabila de tip!
```

Proprietăți induse de tipuri



+ | **Progres** O expresie bine-tipată (căreia i se poate asocia un tip):

- este o **valoare** (nu este o aplicare de funcție) **sau**
- (este aplicarea unei funcții și) poate fi **redușă** (vezi β-redex).

+ | **Conservare** Evaluarea unei expresii bine-tipate produce o expresie **bine-tipată** – de obicei, cu același tip.

- dacă **sinteză de tip** pentru expresia *E* dă tipul *t*, atunci după reducere, valoarea expresiei *E* va fi de tipul *t*.

## Exemple de sinteză de tip

Câteva reguli simplificcate de sinteză de tip



- Formă:  $\frac{\text{premise-1} \dots \text{premise-m}}{\text{concluzie-1} \dots \text{concluzie-n}}$  (nume)

---

- Funcție:  $\frac{\text{Var} :: a \quad \text{Expr} :: b}{\backslash \text{Var} \rightarrow \text{Expr} :: a \rightarrow b}$  (TLambda)
- Aplicație:  $\frac{\text{Expr1} :: a \rightarrow b \quad \text{Expr2} :: a}{(\text{Expr1 Expr2}) :: b}$  (TApp)
- Operatorul +:  $\frac{\text{Expr1} :: \text{Int} \quad \text{Expr2} :: \text{Int}}{\text{Expr1} + \text{Expr2} :: \text{Int}}$  (T+)
- Literalii întregi:  $\frac{}{0, 1, 2, \dots :: \text{Int}}$  (TInt)

Tipare Sintează de tip Tipuri în Haskell TDA 7:10

## Exemple de sinteză de tip

O funcție ne-tipabilă



### Exemplul 3

$$1 \text{ f x} = (\text{x x})$$

$$\frac{x :: a \quad (x x) :: b}{f :: a \rightarrow b} \text{ (TLambda)}$$

$$\frac{x :: c \rightarrow d \quad x :: c}{(x x) :: d} \text{ (TApp)}$$

Ecuția  $c \rightarrow d = c$  nu are soluție (≠ tipuri recursive)  
 ⇒ funcția nu poate fi tipată.

Tipare Sintează de tip Tipuri în Haskell TDA 7:13

## Unificare

Exemplu



### Exemplu

- Pentru a unifica expresiile de tip:
  - $t1 = (a, [b])$
  - $t2 = (\text{Int}, c)$
- putem avea substituțiile (variante):
  - $S1 = \{a \leftarrow \text{Int}, b \leftarrow \text{Int}, c \leftarrow [\text{Int}]\}$
  - $S2 = \{a \leftarrow \text{Int}, c \leftarrow [b]\}$
- Forme comune pentru  $S1$  respectiv  $S2$ :
  - $t1/S1 = t2/S1 = (\text{Int}, [\text{Int}])$
  - $t1/S2 = t2/S2 = (\text{Int}, [b])$

+ **Most general unifier – MGU** Cea mai generală substituție sub care formulele unifică. Exemplu:  $S2$ .

Tipare Sintează de tip Tipuri în Haskell TDA 7:16

## Exemple de sinteză de tip

Transformare de funcție



### Exemplul 1

$$1 \text{ f g} = (\text{g 3}) + 1$$

$$\frac{g :: a \quad (\text{g 3}) + 1 :: b}{f :: a \rightarrow b} \text{ (TLambda)}$$

$$\frac{(\text{g 3}) :: \text{Int} \quad 1 :: \text{Int}}{(\text{g 3}) + 1 :: \text{Int}} \text{ (T+)}$$

$$\Rightarrow b = \text{Int}$$

$$\frac{g :: c \rightarrow d \quad 3 :: c}{(\text{g 3}) :: d} \text{ (TApp)}$$

$$\Rightarrow a = c \rightarrow d, c = \text{Int}, d = \text{Int}$$

$$\Rightarrow f :: (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int}$$

Tipare Sintează de tip Tipuri în Haskell TDA 7:11

## Unificare

Definiție



- la baza sintezei de tip: **unificarea** → legarea variabilelor în timpul procesului de sinteză, în scopul **unificării** diverselor formule de tip elaborate.

+ **Unificare** Procesul de identificare a valorilor **variabilelor** din 2 sau mai multe formule, astfel încât **substituirea** variabilelor prin valorile asociate să conducă la **coincidența** formulelor.

+ **Substituție** O substituție este o mulțime de **legări** variabilă - valoare.

Tipare Sintează de tip Tipuri în Haskell TDA 7:14

## Tip principal

Exemplu și definiție



### Exemplu

- Tipurile:  $t1 = (a, [b])$ ,  $t2 = (\text{Int}, c)$ 
  - MGU:  $S = \{a \leftarrow \text{Int}, c \leftarrow [b]\}$
  - Tipuri mai particulare (instanțe):  $(\text{Integer}, [\text{Integer}]), (\text{Integer}, [\text{Char}]),$  etc
- Funcția:  $\backslash x \rightarrow x$ 
  - Tipuri corecte:  $\text{Int} \rightarrow \text{Int}$ ,  $\text{Bool} \rightarrow \text{Bool}$ ,  $a \rightarrow a$

+ **Tip principal al unei expresii** – Cel mai general tip care descrie complet natura expresiei. Se obține prin utilizarea MGU.

Tipare Sintează de tip Tipuri în Haskell TDA 7:17

## Exemple de sinteză de tip

Combinator de punct fix



### Exemplul 2

$$1 \text{ fix f} = f (\text{fix f})$$

$$\frac{f :: a \quad f (\text{fix f}) :: b}{\text{fix} :: a \rightarrow b} \text{ (TLambda)}$$

$$\frac{f :: c \rightarrow d \quad (\text{fix f}) :: c}{(f (\text{fix f})) :: d} \text{ (TApp)}$$

$$\Rightarrow a = c \rightarrow d, b = d$$

$$\frac{\text{fix} :: e \rightarrow g \quad f :: e}{(\text{fix f}) :: g} \text{ (TApp)}$$

$$\Rightarrow a \rightarrow b = e \rightarrow g, a = e, b = g, c = g$$

$$\Rightarrow \text{fix} :: (c \rightarrow d) \rightarrow b = (g \rightarrow g) \rightarrow g$$

Tipare Sintează de tip Tipuri în Haskell TDA 7:12

## Unificare

Condiții



- O **variabilă de tip** a unifică cu o **expresie de tip** E doar dacă:
  - $E = a$  sau
  - $E \neq a$  și E nu conține a (*occurrence check*).  
Exemplu: a unifică cu  $b \rightarrow c$  dar nu cu  $a \rightarrow b$ .

- 2 **constante** de tip unifică doar dacă sunt egale;

- 2 **aplicații** de tip unifică doar dacă implică același constructor de tip și argumente ce unifică recursiv.

Tipare Sintează de tip Tipuri în Haskell TDA 7:15

TDA

Tipare Sintează de tip Tipuri în Haskell TDA 7:18

## Constructorul de tip Natural

Exemplu de definire TDA 1



### Exemplu

```
1 data Natural = Zero
2             | Succ Natural
3   deriving (Show, Eq)
4
5 unu         = Succ Zero
6 doi         = Succ unu
7
8 addNat Zero n = n
9 addNat (Succ m) n = Succ (addNat m n)
```

Tipare Sinteză de tip Tipuri în Haskell TDA 7:19

## Constructorul de tip Pair

Comentarii



- Constructor de tip: Pair
  - polimorfic, binar;
  - generează un tip în momentul aplicării asupra 2 tipuri.

- Constructor de date: P, binar:

```
1 P :: a -> b -> Pair a b
```

Tipare Sinteză de tip Tipuri în Haskell TDA 7:22

## Polimorfism



+ **Polimorfism parametric** Manifestarea **aceluiași** comportament pentru parametri de tipuri **diferite**. Exemplu: id, Pair.

+ **Polimorfism ad-hoc** Manifestarea unor comportamente **diferite** pentru parametri de tipuri **diferite**. Exemplu: ==.

Motivatie Clase Haskell Clase în Haskell Aplicații clase 8:3

## Constructorul de tip Natural

Comentarii



- Constructor de tip: Natural
  - nular;
  - **se confundă** cu tipul pe care-l construiește.

- Constructori de date:
  - Zero: nular
  - Succ: unar

- Constructorii de date ca **funcții**, dar utilizabile în *pattern matching*.

```
1 Zero :: Natural
2 Succ :: Natural -> Natural
```

Tipare Sinteză de tip Tipuri în Haskell TDA 7:20

## Cursul 8: Clase în Haskell



### 1 Motivație

### 2 Clase Haskell

### 3 Aplicații ale claselor

## Motivație

Motivatie Clase Haskell Clase în Haskell Aplicații clase 8:1

## Motivație

Exemplu



### Exemplu

Să se definească operația show, capabilă să producă reprezentarea oricărui obiect ca șir de caractere. Comportamentul este **specific** fiecărui tip (polimorfism **ad-hoc**).

```
1 show 3 → "3"
2 show True → "True"
3 show 'a' → "'a'"
4 show "a" → "\"a\""
```

Motivatie Clase Haskell Clase în Haskell Aplicații clase 8:4

## Constructorul de tip Pair

Exemplu de definire TDA 2



### Exemplu

```
1 data Pair a b = P a b
2   deriving (Show, Eq)
3
4 pair1         = P 2 True
5 pair2         = P 1 pair1
6
7 myFst (P x y) = x
8 mySnd (P x y) = y
```

Tipare Sinteză de tip Tipuri în Haskell TDA 7:21

## Motivație

Varianta 1 – Funcții dedicate fiecărui tip



```
1 showBool True = "True"
2 showBool False = "False"
3
4 showChar c = ' ' ++ [c] ++ ' '
5
6 showString s = "\" " ++ s ++ "\" "
```

Motivatie Clase Haskell Clase în Haskell Aplicații clase 8:5

## Motivație

Varianta 1 – Funcții dedicate – discuție



- Dorim să implementăm funcția `showNewLine`, care adaugă caracterul "linie nouă" la reprezentarea ca șir:

```
1 showNewLine x = (show...? x) ++ "\n"
```

- `showNewLine` **nu** poate fi polimorfică ⇒ avem nevoie de `showNewLineBool`, `showNewLineChar` etc.

- Alternativ, trimiterea ca **parametru** a funcției `show*` corespunzătoare:

```
1 showNewLine sh x = (sh x) ++ "\n"
2 showNewLineBool = showNewLine showBool
```

- **Prea general**, fiind posibilă trimiterea unei funcții cu alt comportament, în măsura în care respectă tipul.

Motivatie      Clase Haskell  
Clase în Haskell      Aplicații clase      8 : 6

## Motivație

Varianta 2 – Supraîncărcare – discuție (1)



- Ce **tip** au funcțiile `show`, respectiv `showNewLine`?

```
1 show      :: Show a => a -> String
2 showNewLine :: Show a => a -> String
```

Semnificație: *Dacă tipul a este membru al clasei Show, (i.e. funcția show este definită pe valorile tipului a), atunci funcțiile au tipul a -> String.*

- **Context**: constrângeri suplimentare asupra variabilelor din tipul funcției:

```
Show a =>
context
```

- **Propagarea** constrângerilor din contextul lui `show` către contextul lui `showNewLine`.

Motivatie      Clase Haskell  
Clase în Haskell      Aplicații clase      8 : 9

## Clase Haskell vs. Clase în POO



### Haskell

- **Tipurile** sunt mulțimi de **valori**;
- **Clasele** sunt mulțimi de **tipuri**; tipurile *aderă* la clase;
- **Instanțierea** claselor de către tipuri pentru ca funcțiile definite în clasă să fie disponibile pentru valorile tipului;
- Operațiile specifice clasei sunt implementate în cadrul declarației de instanțiere.

### POO (e.g. Java)

- **Clasele** sunt mulțimi de **obiecte** (*instanțe*);
- **Interfețele** sunt mulțimi de **clase**; clasele *implementează* interfețe;
- **Implementarea** interfețelor de către clase pentru ca funcțiile definite în interfață să fie disponibile pentru instanțele clasei;
- Operațiile specifice interfeței sunt implementate în cadrul definiției clasei.

Motivatie      Clase Haskell  
Clase în Haskell      Aplicații clase      8 : 12

## Motivație

Cum putem obține un comportament coerent?



- Într-un limbaj care suportă supraîncărcarea operatorilor / funcțiilor, aș defini câte o funcție `show` pentru fiecare tip care suportă afișare (cum este `toString` în Java)

- dar cum pot defini în mod coerent tipul lui `showNewLine`?

"`showNewLine` poate primi ca argument orice tip a supraîncărcat funcția `show`."

⇒ **Clasa** (*mulțimea de tipuri*) `Show`, care necesită implementarea funcției `show`.

Motivatie      Clase Haskell  
Clase în Haskell      Aplicații clase      8 : 7

## Motivație

Varianta 2 – Supraîncărcare – discuție



- Contexte utilizabile și la **instanțiere**:

```
1 instance (Show a, Show b) => Show (a, b) where
2   show (x, y) = "(" ++ (show x)
3               ++ ",_" ++ (show y)
4               ++ ")"
```

- Tipul *pereche* reprezentabil ca șir doar dacă tipurile celor doi membri respectă **aceeași** proprietate (dată de contextul `Show`).

Motivatie      Clase Haskell  
Clase în Haskell      Aplicații clase      8 : 10

## Clase și instanțe

Definiții



+ | **Clasa** – Mulțime de tipuri ce pot supraîncarca operațiile specifice clasei. Reprezintă o modalitate structurată de control asupra polimorfismului *ad-hoc*. Exemplu: clasa `Show`, cu operația `show`.

+ | **Instanță a unei clase** – Tip care supraîncarcă operațiile clasei.

Exemplu: tipul `Bool` în raport cu clasa `Show`.

- *clasa* definește funcțiile **suportate**;
- clasa se definește peste o variabilă care stă pentru **constructorul unui tip**;
- *instanța* definește **implementarea** funcțiilor.

Motivatie      Clase Haskell  
Clase în Haskell      Aplicații clase      8 : 13

## Motivație

Varianta 2 – Supraîncărcarea funcției → funcție polimorfică ad-hoc



- Definirea **mulțimii** `Show`, a **tipurilor** care expun `show`

```
1 class Show a where
2   show :: a -> String
```

- Precizarea **apartenenței** unui tip la această mulțime (instanța *aderă* la clasă)

```
1 instance Show Bool where
2   show True  = "True"
3   show False = "False"
4 instance Show Char where
5   show c = "'" ++ [c] ++ "'"
```

⇒ Funcția `showNewLine` **polimorfică!**

```
1 showNewLine x = show x ++ "\n"
```

Motivatie      Clase Haskell  
Clase în Haskell      Aplicații clase      8 : 8

## Clase Haskell

## Clase predefinite

`Show`, `Eq`



```
1 class Show a where
2   show :: a -> String
3
4 class Eq a where
5   (==), (/=) :: a -> a -> Bool
6   x /= y     = not (x == y)
7   x == y     = not (x /= y)
```

- Posibilitatea scrierii de definiții **implicit**e (v. liniile 6–7).
- Necesitatea suprascrierii **cel puțin unuia** din cei 2 operatori ai clasei `Eq` pentru instanțierea corectă.

Motivatie      Clase Haskell  
Clase în Haskell      Aplicații clase      8 : 14

## Clase predefinite

Ord



```
1 class Eq a => Ord a where
2   (<), (<=), (>=), (>) :: a -> a -> Bool
3   ...
```

- contextele – utilizabile și la **definirea** unei clase.
- clasa Ord **moștenește** clasa Eq, cu preluarea operațiilor din clasa moștenită.
- este **necesară** aderarea la clasa Eq în momentul instanțierii clasei Ord.
- este **suficientă** supradefinirea lui (<=) la instanțiere.

Motivatie Clase Haskell  
Clase în Haskell Aplicații clase 8 : 15

## invert Problemă



**invert**  
Fie constructorii de tip:

```
1 data Pair a = P a a
2
3 data NestedList a
4   = Atom a
5   | List [NestedList a]
```

Să se definească operația **invert**, aplicabilă pe valori de tipuri diferite, inclusiv Pair a și NestedList a, comportamentul fiind **specific** fiecărui tip.

Motivatie Clase Haskell  
Clase în Haskell Aplicații clase 8 : 18

## contents Varianta 1a



```
1 class Container a where
2   contents :: a -> [a]
3 instance Container [x] where
4   contents = id
```

Testăm pentru contents [1,2,3]:

- Conform definiției clasei:  

```
1 contents :: Container [a] => [a] -> [[a]]
```
- Conform supraîncărcării funcției (id):  

```
1 contents :: Container [a] => [a] -> [a]
```
- Ecuația  $[a] = [[a]]$  **nu are soluție** ⇒ **eroare**.

Motivatie Clase Haskell  
Clase în Haskell Aplicații clase 8 : 21

## Utilizarea claselor predefinite

Pentru tipuri de date noi



- **Anumite** tipuri de date (definite folosind data) pot beneficia de implementarea **automată** a anumitor funcționalități, oferite de tipurile predefinite în Prelude:

- Eq, Read, Show, Ord, Enum, Ix, Bounded.

```
1 data Alarm = Soft | Loud | Deafening
2   deriving (Eq, Ord, Show)
```

- variabilele de tipul Alarm pot fi comparate, testate la egalitate, și afișate.

Motivatie Clase Haskell  
Clase în Haskell Aplicații clase 8 : 16

## invert Implementare



```
1 class Invertible a where
2   invert :: a -> a
3   invert = id
4
5 instance Invertible (Pair a) where
6   invert (P x y) = P y x
7 instance Invertible a => Invertible (NestedList a) where
8   invert (Atom x) = Atom (invert x)
9   invert (List x) = List $ reverse $ map invert x
10 instance Invertible a => Invertible [a] where
11   invert lst = reverse $ map invert lst
12 instance Invertible Int ...
```

- Necesitatea **contextului**, în cazul tipurilor [a] și NestedList a, pentru inversarea elementelor **înselor**.

Motivatie Clase Haskell  
Clase în Haskell Aplicații clase 8 : 19

## contents Varianta 1b



```
1 class Container a where
2   contents :: a -> [b]
3 instance Container [x] where
4   contents = id
```

Testăm pentru contents [1,2,3]:

- Conform definiției clasei:  

```
1 contents :: Container [a] => [a] -> [b]
```
- Conform supraîncărcării funcției (id):  

```
1 contents :: Container [a] => [a] -> [a]
```
- Ecuația  $[a] = [b]$  **are** soluție pentru  $a = b$ , dar tipul  $[a] -> [a]$  este **insuficient** de general (prea specific) în raport cu  $[a] -> [b] ⇒ eroare!$

Motivatie Clase Haskell  
Clase în Haskell Aplicații clase 8 : 22

## Aplicații ale claselor

## contents Problemă



**contents**

Să se definească operația **contents**, aplicabilă pe obiecte **structurate**, inclusiv pe cele aparținând tipurilor Pair a și NestedList a, care întoarce elementele din componentă, sub forma unei **liste** Haskell.

```
1 class Container a where
2   contents :: a -> [...?]
```

- a este tipul unui **container**, e.g. NestedList b
- Elementele listei întoarse sunt cele **din container**
- Cum **precizăm** tipul acestora (b)?

Motivatie Clase Haskell  
Clase în Haskell Aplicații clase 8 : 20

## contents Varianta 2



**Soluție** clasa primește **constructorul** de tip, și nu tipul container propriu-zis (rezultat după aplicarea constructorului) ⇒ includem tipul conținut de container în expresia de tip a funcției contents:

```
1 class Container t where
2   contents :: t a -> [a]
3
4 instance Container Pair where
5   contents (P x y) = [x, y]
6
7 instance Container NestedList where
8   contents (Atom x) = [x]
9   contents (Seq x) = concatMap contents x
10
11 instance Container [] where contents = id
```

Motivatie Clase Haskell  
Clase în Haskell Aplicații clase 8 : 23



```

1 fun1 :: Eq a => a -> a -> a -> a
2 fun1 x y z = if x == y then x else z
3
4 fun2 :: (Container a, Invertible (a b),
5 Eq (a b)) => (a b) -> (a b) -> [b]
6 fun2 x y = if (invert x) == (invert y)
7           then contents x else contents y
8
9 fun3 :: Invertible a => [a] -> [a] -> [a]
10 fun3 x y = (invert x) ++ (invert y)
11
12 fun4 :: Ord a => a -> a -> a -> a
13 fun4 x y z = if x == y then z else
14             if x > y then x else y

```

### Caracteristici ale paradigmelor de programare



- În principal, paradigma este definită de
  - elementele principale din sintaxa limbajului – e.g. existența și semnificația **variabilelor**, semnificația **operatorilor** asupra datelor, modul de construire a programului;
  - modul de construire al **tipurilor** variabilelor;
  - modul de definire și statutul **operatorilor** – elementele principale de prelucrare a datelor din program (e.g. obiecte, funcții, predicate);
  - **legarea** variabilelor, efecte laterale, transparentă referențială, modul de transfer al parametrilor pentru elementele de prelucrare a datelor.



- **Simplificarea** contextului lui fun3, de la Invertible [a] la Invertible a.
- **Simplificarea** contextului lui fun4, de la (Eq a, Ord a) la Ord a, din moment ce clasa Ord este **derivată** din clasa Eq.



- **Paradigma de programare** – un mod de a:
  - aborda rezolvarea unei probleme printr-un program;
  - structura un program;
  - reprezenta datele dintr-un program;
  - implementa diversele aspecte dintr-un program (**cum** prelucrăm datele);
- Un limbaj poate include caracteristici din una sau mai multe paradigme;
  - în general există o paradigmă dominantă;
- **Atenție!** Paradigma nu are legătură cu sintaxa limbajului!

### Variabile și valori de prim rang



- 34 Caracteristici ale paradigmelor de programare
- 35 Variabile și valori de prim rang
- 36 Tipare a variabilelor
- 37 Legarea variabilelor
- 38 Modul de evaluare



- paradigmele sunt legate teoretic de o **mașină de calcul** în care prelucrările caracteristice paradigmei se fac la nivelul mașinii;
- **dar** putem executa orice program, scris în orice paradigmă, pe orice mașină.



- în majoritatea limbajelor există variabile, ca **NUME** date unor valori – rezultatul anumitor procesări (calcul, inferențe, substituții);
- variabilele pot fi o **referință** pentru un spațiu de memorie sau pentru un rezultat abstract;
- elementele de procesare a datelor pot sau nu să fie **valori de prim rang** (să poată fi asociate cu variabile).

## Funcții ca valori de prim rang

Definiție

APP

+ **Valoare de prim rang** – O valoare care poate fi:

- creată dinamic
- stocată într-o variabilă
- trimisă ca parametru unei funcții
- întoarsă dintr-o funcție

Ex) Să se scrie funcția `compose`, ce primește ca parametri alte 2 funcții, `f` și `g`, și întoarce funcția obținută prin compunerea lor, `f ∘ g`.

Caracteristici Variabile & valori Tipare Legarea variabilelor Evaluare 9 : 8  
Concluzie – Paradigma Funcțională

## Funcții ca valori de prim rang: Compose

Racket & Haskell

APP

• Racket:

```
1 (define compose
2   (lambda (f g)
3     (lambda (x)
4       (f (g x))))))
```

• Haskell:

```
1 compose = (.)
```

• În Racket și Haskell, funcțiile **sunt** valori de prim rang.

• mai mult, ele pot fi **aplicate parțial**, și putem avea **funcționale** – funcții care iau alte funcții ca parametru.

Caracteristici Variabile & valori Tipare Legarea variabilelor Evaluare 9 : 11  
Concluzie – Paradigma Funcțională

## Tipare statică vs. dinamică

Exemplu

APP

Exemplu Tipare dinamică

JavaScript:  
`var x = 5;`  
`if(condition) x = "here";`  
`print(x);` → ce tip are `x` aici?

Exemplu Tipare statică

Java:  
`int x = 5;`  
`if(condition)`  
  `x = "here";` → **Eroare la compilare**: `x` este `int`.  
`print(x);`

Caracteristici Variabile & valori Tipare Legarea variabilelor Evaluare 9 : 14  
Concluzie – Paradigma Funcțională

## Funcții ca valori de prim rang: Compose

C

APP

```
1 int compose(int (*f)(int), int (*g)(int), int x) {
2   return (*f)((*g)(x));
3 }
```

• în C, funcțiile **nu** sunt valori de prim rang;

• pot scrie o funcție care compune două funcții pe o anumită valoare (ca mai sus)

• pot întoarce pointer la o funcție existentă

• dar nu pot crea o referință (pointer) la o funcție **nouă**, care să fie folosit apoi ca o funcție obișnuită

Caracteristici Variabile & valori Tipare Legarea variabilelor Evaluare 9 : 9  
Concluzie – Paradigma Funcțională

## Tipare a variabilelor

## Tipare statică vs. dinamică

Caracteristici

APP

Tipare statică

- La compilare
- Valori și variabile
- Rulare mai rapidă
- Rigidă: sancționează orice construcție
- Debugging mai facil
- Declarații explicite sau inferențe de tip
- Pascal, C, C++, Java, Haskell

Tipare dinamică

- La rulare
- Doar valori
- Rulare mai lentă (necesită verificarea tipurilor)
- Flexibilă: sancționează doar când este necesar
- Debugging mai dificil
- Permite metaprogramare (v. `eval`)
- Python, Scheme/Racket, Prolog, JavaScript, PHP

Caracteristici Variabile & valori Tipare Legarea variabilelor Evaluare 9 : 15  
Concluzie – Paradigma Funcțională

## Funcții ca valori de prim rang:

Java

APP

```
1 abstract class Func<U, V> {
2   public abstract V apply(U u);
3
4   public <T> Func<T, V> compose(final Func<T, U> f) {
5     final Func<U, V> outer = this;
6
7     return new Func<T, V>() {
8       public V apply(T t) {
9         return outer.apply(f.apply(t));
10      }
11    };
12  }
13 }
```

• În Java, funcțiile **nu** sunt valori de prim rang – pot crea rezultatul dar este complicat, și rezultatul nu este o funcție obișnuită, ci un obiect.

Caracteristici Variabile & valori Tipare Legarea variabilelor Evaluare 9 : 10  
Concluzie – Paradigma Funcțională

## Modalități de tipare

APP

• Rolul tipurilor: exprimare a intenției programatorului, abstractizare, documentare, optimizare, verificare

+ **Tipare** – modul de gestionare a tipurilor.

• Clasificare după momentul verificării:

- statică
- dinamică

• Clasificare după rigiditatea regulilor:

- tare
- slabă

Caracteristici Variabile & valori Tipare Legarea variabilelor Evaluare 9 : 13  
Concluzie – Paradigma Funcțională

## Tipare tare vs. slabă

Exemple

APP

• Clasificare după libertatea de a agrega valori de tipuri diferite.

Exemplu Tipare tare

`1 + "23" → Eroare` (Haskell, Python)

Exemplu Tipare slabă

`1 + "23" = 24` (Visual Basic)  
`1 + "23" = "123"` (JavaScript)

Caracteristici Variabile & valori Tipare Legarea variabilelor Evaluare 9 : 16  
Concluzie – Paradigma Funcțională

## Legarea variabilelor

## Efecte laterale (*side effects*)

Consecințe

- Exemplu În expresia `x-- + ++x`, cu `x = 0`:
  - evaluarea stânga → dreapta produce `0 + 0 = 0`
  - evaluarea dreapta → stânga produce `1 + 1 = 2`
  - dacă înlocuim cele două subexpresii cu valorile pe care le reprezintă, obținem
$$x + (x + 1) = 0 + 1 = 1$$
- Importanța **ordinii de evaluare!**
- Dependențe **implicit**e, puțin lizibile și posibile generatoare de bug-uri.

## Transparență referențială

Pentru funcții

+ **Funcție transparentă referențială**: rezultatul întors depinde **exclusiv** de parametri.

Exemplu

```
int transparent(int x) {
    return x + 1;
}

int opaque(int x) {
    return x + ++g;
}
```

- `opaque(3) - opaque(3) != 0!`
- Funcții transparente: `log`, `sin` etc.
- Funcții opace: `time`, `read` etc.

## Legarea variabilelor

Impactul asupra programului

- două posibilități esențiale:
  - un nume este întotdeauna legat (într-un anumit context) la aceeași valoare / la același calcul ⇒ numele **stă pentru un calcul**;
    - legare **statică**.
  - un nume (aceeași variabilă) poate fi legat la mai multe valori pe parcursul execuției ⇒ numele **stă pentru un spațiu de stocare** – fiecare element de stocare fiind identificat printr-un nume;
    - legare **dinamică**.

## Efecte laterale (*side effects*)

Consecințe asupra programării leneșe

- În prezența efectelor laterale, programarea leneșă devine foarte dificilă;
- Efectele laterale pot fi gestionate corect numai atunci când **secvența** evaluării este garantată → garanție inexistentă în programarea leneșă.
  - nu știm când anume va fi **nevoie** de valoarea unei expresii.

## Transparență referențială

Avantaje

- Lizibilitatea** codului;
- Demonstrarea formală a **corectitudinii** programului – mai ușoară datorită lipsei **stării**;
- Optimizare** prin reordonarea instrucțiunilor de către compilator și prin **caching**;
- Paralelizare** masivă, prin eliminarea modificărilor concurente.

## Efecte laterale (*side effects*)

Definiție

Exemplu În expresia `2 + (i = 3)`, subexpresia `(i = 3)`:

- produce **valoarea** 3, conducând la rezultatul 5 al întregii expresii;
- are **efectul lateral** de inițializare a lui `i` cu 3.

+ **Efect lateral** Pe lângă valoarea pe care o produce, o expresie sau o funcție poate **modifica** starea globală.

- Inerente în situațiile în care programul interacționează cu exteriorul → **I/O!**

## Transparență referențială

Pentru expresii

+ **Transparență referențială** Confundarea unui obiect ("valoare") cu referința la acesta.

+ **Expresie transparentă referențială**: posedă o unică valoare, cu care poate fi substituită, **păstrând** semnificația programului.

Exemplu

- `x-- + ++x` → **nu**, valoarea depinde de ordinea de evaluare
- `x = x + 1` → **nu**, două evaluări consecutive vor produce rezultate diferite
- `x` → ar putea fi, în funcție de statutul lui `x` (globală, statică etc.)

## Modul de evaluare

## Evaluare

Mod de evaluare și execuția programelor

- modul de evaluare al expresiilor dictează modul în care este executat programul;
- este legat de funcționarea **mașinii teoretice** corespunzătoare paradigmei;
- ne interesează în special ordinea în care expresiile se evaluează;
- în final, întregul program se evaluează la o valoare;
- important în modul de evaluare este modul de **evaluare / transfer a parametrilor**.

Caracteristici Variabile & valori Tipare Legarea variabilelor Evaluare 9 : 26  
Concluzie – Paradigma Funcțională

## Call by sharing

În evaluarea aplicativă

- Variantă a *call by value*;
- Trimiterea unei **referințe** la obiect;
- Modificări locale asupra **referinței** invizibile la apelant;
- Modificări locale asupra **obiectului** referit vizibile la apelant;
- Racket, Java;

Caracteristici Variabile & valori Tipare Legarea variabilelor Evaluare 9 : 29  
Concluzie – Paradigma Funcțională

## Call by need

În evaluarea normală

- Variantă a *call by name*;
- Evaluarea unui parametru doar la **prima** utilizare a acestuia;
- **Memorarea** valorii unui parametru deja evaluat și returnarea acesteia în cazul utilizării repetate a aceluiași parametru (datorită transparenței referențiale, o aceeași expresie are întotdeauna aceeași valoare) – **memoizare**;
- în Haskell.

Caracteristici Variabile & valori Tipare Legarea variabilelor Evaluare 9 : 32  
Concluzie – Paradigma Funcțională

## Transferul parametrilor

- Evaluare **aplicativă** – parametrii sunt evaluați înainte de evaluarea corpului funcției.
  - *Call by value*
  - *Call by sharing*
  - *Call by reference*
- Evaluare **normală** – funcția este evaluată fără ca parametrii să fie evaluați înainte.
  - *Call by name*
  - *Call by need*

Caracteristici Variabile & valori Tipare Legarea variabilelor Evaluare 9 : 27  
Concluzie – Paradigma Funcțională

## Call by reference

În evaluarea aplicativă

- Trimiterea unei **referințe** la obiect;
- Modificări locale asupra referinței și obiectului referit **vizibile** la apelant;
- Folosirea “&” în C++.

Caracteristici Variabile & valori Tipare Legarea variabilelor Evaluare 9 : 30  
Concluzie – Paradigma Funcțională

## Cursul 10: Logica cu predicate de ordinul I

- 39 Logica propozițională
- 40 Evaluarea valorii de adevăr
- 41 Logica cu predicate de ordinul întâi
- 42 LPOI – Semantică
- 43 Forme normale
- 44 Unificare și rezoluție

Logica propozițională Evaluare LPOI LPOI – Semantică Forme normale Unificare și rezoluție 10 : 1  
Logica cu predicate de ordinul I

## Call by value

În evaluarea aplicativă

### Exemplu

```
1 // C sau Java                                1 // C
2 void f(int x) {                               2 void g(struct str s) {
3     x = 3;                                    3     s.member = 3;
4 }                                             4 }
```

· Efectul liniilor 3 este **invizibil** la apelant.

- Evaluarea parametrilor **înaintea** aplicației funcției și transferul unei **copii** a valorii acestuia
- Modificări locale **invizibile** la apelant
- C, C++, tipurile primitive Java

Caracteristici Variabile & valori Tipare Legarea variabilelor Evaluare 9 : 28  
Concluzie – Paradigma Funcțională

## Call by name

În evaluarea normală

- Argumente **neevaluate** în momentul aplicării funcției → substituție directă (textuală) în corpul funcției;
- Evaluare parametrilor la cerere, de **fiecare** dată când este nevoie de valoarea acestora;
- în calculul  $\lambda$ .

Caracteristici Variabile & valori Tipare Legarea variabilelor Evaluare 9 : 31  
Concluzie – Paradigma Funcțională

## Logică

- formalism simbolic pentru reprezentarea faptelor și raționament.
- se bazează pe ideea de **valoare de adevăr** – e.g. *Adevărat* sau *Fals*.
- permite realizarea de argumente (argumentare) și demonstrații – deducție, inducție, rezoluție, etc.

Logica propozițională Evaluare LPOI LPOI – Semantică Forme normale Unificare și rezoluție 10 : 2  
Logica cu predicate de ordinul I

# Logica propozițională

## Logica propozițională Semantică

- Scop: dezvoltarea unor mecanisme de prelucrare, aplicabile **independent** de valoarea de adevăr a propozițiilor într-o situație particulară.
- Accent pe **relațiile** între propozițiile compuse și cele constituente.
- Pentru explicitarea propozițiilor → utilizarea conceptului de **interpretare**.

## Semantică Propoziții compuse (2)

- **Implicatie:**  $(\alpha \Rightarrow \beta)^I = \begin{cases} false & \text{dacă } \alpha^I = true \text{ și } \beta^I = false \\ true & \text{altfel} \end{cases}$
- **Echivalență:**  $(\alpha \Leftrightarrow \beta)^I = \begin{cases} true & \text{dacă } \alpha \Rightarrow \beta \wedge \beta \Rightarrow \alpha \\ false & \text{altfel} \end{cases}$

## Logica propozițională Context și elemente principale

- Cadru pentru:
  - **descrierea** proprietăților obiectelor, prin intermediul unui **limbaj**, cu o **semantică** asociată;
  - **deducerea** de noi proprietăți, pe baza celor existente.
- Expresia din limbaj: **propoziția**, corespunzătoare unei afirmații, ce poate fi adevărată sau falsă.
- Exemplu: "Afară este frumos."
- Accepții asupra unei propoziții:
  - **secvența de simboluri** utilizate sau
  - **înțelesul** propriu-zis al acesteia, într-o **interpretare**.

## Semantică Interpretare

+ | **Interpretare** Multime de **asocieri** între fiecare propoziție **simplă** din limbaj și o valoare de adevăr.

Exemplu

- |                  |                  |
|------------------|------------------|
| Interpretarea I: | Interpretarea J: |
| ● $p^I = false$  | ● $p^J = true$   |
| ● $q^I = true$   | ● $q^J = true$   |
| ● $r^I = false$  | ● $r^J = true$   |
- cum știu dacă  $p$  este adevărat sau fals? Pot ști dacă știu **interpretarea** –  $p$  este doar un **nume** pe care îl dau unei propoziții concrete.

## Evaluarea valorii de adevăr

## Logica propozițională Sintaxă

- 2 categorii de propoziții
  - simple → fapte **atomice**: "Afară este frumos."
  - compuse → **relații** între propoziții mai simple: "Telefonul sună și câinele latră."
- Propoziții simple:  $p, q, r, \dots$
- Negatii:  $\neg \alpha$
- Conjunții:  $(\alpha \wedge \beta)$
- Disjunții:  $(\alpha \vee \beta)$
- Implicații:  $(\alpha \Rightarrow \beta)$
- Echivalențe:  $(\alpha \Leftrightarrow \beta)$

## Semantică Propoziții compuse (1)

- Sub o interpretare **fixată** → **dependența** valorii de adevăr a unei propoziții compuse de valorile de adevăr ale celor constituente
- **Negatie:**  $(\neg \alpha)^I = \begin{cases} true & \text{dacă } \alpha^I = false \\ false & \text{altfel} \end{cases}$
- **Conjunție:**  $(\alpha \wedge \beta)^I = \begin{cases} true & \text{dacă } \alpha^I = true \text{ și } \beta^I = true \\ false & \text{altfel} \end{cases}$
- **Disjuncție:**  $(\alpha \vee \beta)^I = \begin{cases} false & \text{dacă } \alpha^I = false \text{ și } \beta^I = false \\ true & \text{altfel} \end{cases}$

## Evaluare Cum determinăm valoarea de adevăr?

+ | **Evaluare** Determinarea **valorii de adevăr** a unei **propoziții**, sub o **interpretare**, prin aplicarea regulilor semantice anterioare.

Exemplu

- Interpretarea I:
  - $p^I = false$
  - $q^I = true$
  - $r^I = false$
- Propoziția:  $\phi = (p \wedge q) \vee (q \Rightarrow r)$   
 $\phi^I = (false \wedge true) \vee (true \Rightarrow false) = false \vee false = false$

## Valoarea de adevăr în afara interpretării

Satisfiabilitate, Validitate, Nesatisfiabilitate

$P \vee \bar{P}$

+ **Satisfiabilitate** Proprietatea unei propoziții care este adevărată sub cel puțin o interpretare. Acea interpretare **satisface** propoziția.

+ **Validitate** Proprietatea unei propoziții care este adevărată în toate interpretările. Propoziția se mai numește **tautologie**.

Exemplu Propoziția  $p \vee \neg p$  este **validă**.

+ **Nesatisfiabilitate** Proprietatea unei propoziții care este falsă în toate interpretările. Propoziția se mai numește **contradicție**.

Exemplu Propoziția  $p \wedge \neg p$  este **nesatisfiabilă**.

## Derivabilitate

Verificare

$P \vee \bar{P}$

- Verificabilă prin metoda tabeli de adevăr: **toate** intrările pentru care **premisele** sunt adevărate trebuie să inducă adevărul **concluziei**.

Demonstrăm că  $\{p, p \Rightarrow q\} \models q$ .

| $p$   | $q$   | $p \Rightarrow q$ |
|-------|-------|-------------------|
| true  | true  | true              |
| true  | false | false             |
| false | true  | true              |
| false | false | true              |

Singura intrare în care ambele premise,  $p$  și  $p \Rightarrow q$ , sunt adevărate, precizează și adevărul concluziei,  $q$ .

## Inferență

Definiție

$P \vee \bar{P}$

+ **Inferență** – Derivarea **mecanică** a concluziilor unui set de premise.

+ **Regulă de inferență** – **Procedură** de calcul capabilă să deriveze concluziile unui set de premise. Derivabilitatea mecanică a concluziei  $\phi$  din mulțimea de premise  $\Delta$ , utilizând **regula de inferență** *inf*, se notează  $\Delta \vdash_{inf} \phi$ .

Exemplu Modus Ponens (MP) : 
$$\frac{\alpha \Rightarrow \beta \quad \alpha}{\beta}$$

Exemplu Modus Tollens : 
$$\frac{\alpha \Rightarrow \beta \quad \neg \beta}{\neg \alpha}$$

## Valoarea de adevăr în afara interpretării

Metoda tabeli de adevăr

$P \vee \bar{P}$

Exemplu Metoda tabeli de adevăr

| $p$   | $q$   | $r$   | $(p \wedge q) \vee (q \Rightarrow r)$ |
|-------|-------|-------|---------------------------------------|
| true  | true  | true  | true                                  |
| true  | true  | false | true                                  |
| true  | false | true  | true                                  |
| true  | false | false | true                                  |
| false | true  | true  | true                                  |
| false | true  | false | false                                 |
| false | false | true  | false                                 |
| false | false | false | false                                 |

$\Rightarrow$  Propoziția  $(p \wedge q) \vee (q \Rightarrow r)$  este **satisfiabilă**.

## Derivabilitate

Formulări echivalente

$P \vee \bar{P}$

- $\{\phi_1, \dots, \phi_n\} \models \phi$

sau

- Propoziția  $\phi_1 \wedge \dots \wedge \phi_n \Rightarrow \phi$  este **validă**

sau

- Propoziția  $\phi_1 \wedge \dots \wedge \phi_n \wedge \neg \phi$  este **nesatisfiabilă**

## Inferență

Proprietăți ale regulilor

$P \vee \bar{P}$

+ **Consistență (soundness)** – Regula de inferență determină **numai** propoziții care sunt, într-adevăr, **consecințe logice** ale premiselor.

$\Delta \vdash_{inf} \phi \Rightarrow \Delta \models \phi$ .

+ **Completitudine (completeness)** – Regula de inferență determină **toate consecințele logice** ale premiselor.  $\Delta \models \phi \Rightarrow \Delta \vdash_{inf} \phi$ .

- Ideal, **ambele** proprietăți – “nici în plus, nici în minus” –  $\Delta \models \phi \Leftrightarrow \Delta \vdash_{inf} \phi$
- Incompletitudinea** regulii *Modus Ponens*, din imposibilitatea scrierii oricărei propoziții ca implicație.

## Derivabilitate

Definiție

$P \vee \bar{P}$

+ **Derivabilitate logică** Proprietatea unei propoziții de a reprezenta **consecința logică** a unei mulțimi de alte propoziții, numite **premise**. Mulțimea de propoziții  $\Delta$  derivă propoziția  $\phi$  ( $\Delta \models \phi$ ) dacă și numai dacă **orice** interpretare care satisface toate propozițiile din  $\Delta$  satisface și  $\phi$ .

Exemplu

- $\{p\} \models p \vee q$
- $\{p, q\} \models p \wedge q$
- $\{p\} \not\models p \wedge q$
- $\{p, p \Rightarrow q\} \models q$

## Inferență

Motivație

$P \vee \bar{P}$

- Cresterea **exponențială** a numărului de interpretări în raport cu numărul de propoziții simple.
- De aici, **diminuarea** valorii practice a metodelor **semantice**, precum cea a tabeli de adevăr.
- Alternativ, metode **sintactice**, care manipulează doar reprezentarea simbolică.
  - Inferență  $\rightarrow$  Derivare **mecanică**  $\rightarrow$  demers de **calcul**, în scopul verificării derivabilității logice.
  - folosind **metodele de inferență**, putem construi o **mașină de calcul**.

Logica cu predicate de ordinul întâi

- **Extensie** a logicii propoziționale, cu explicitarea:
  - **obiectelor** din universul problemei;
  - **relațiilor** dintre acestea.
- Logica propozițională:
  - $p$ : “Andrei este prieten cu Bogdan.”
  - $q$ : “Bogdan este prieten cu Andrei.”
  - $p \Leftrightarrow q$  – pot ști doar din interpretare.
 → **Opacitate** în raport cu obiectele și relațiile referite.
- FOPL:
  - Generalizare:  $prieten(x, y)$ : “ $x$  este prieten cu  $y$ .”
  - $\forall x, \forall y. (prieten(x, y) \Leftrightarrow prieten(y, x))$
  - Aplicare pe cazuri **particulare**.
  - **Transparentă** în raport cu obiectele și relațiile referite.

+ **Atomi** (relații): atomul  $p(t_1, \dots, t_n)$ , unde  $p$  este un **predicat**  $n$ -ar și  $t_1, \dots, t_n$  sunt termeni.

Exemple

- $impar(3)$
- $varsta(ion, 20)$
- $= (+ (2, 3), 5)$

LPOI – Semantică

- + **Constante** – obiecte particulare din universul discursului:  $c, d, andrei, bogdan, \dots$
- + **Variabile** – obiecte generice:  $x, y, \dots$
- + **Simboluri funcționale** –  $succesor, +, abs \dots$
- + **Simboluri relaționale (predicate)** – relații  $n$ -are peste obiectele din universul discursului:  $prieten = \{(andrei, bogdan), (bogdan, andrei), \dots\}, impar = \{1, 3, \dots\}, \dots$
- + **Conectori logici**  $\neg, \wedge, \vee, \Rightarrow, \Leftarrow$
- + **Cuantificatori**  $\forall, \exists$

+ **Propoziții** (fapte) – dacă  $x$  variabilă,  $A$  atom, și  $\alpha$  și  $\beta$  propoziții, atunci o propoziție are forma:

- Fals, Adevărat:  $\perp, \top$
- **Atomi**:  $A$
- **Negații**:  $\neg \alpha$
- **Conectori**:  $\alpha \wedge \beta, \alpha \Rightarrow \beta, \dots$
- **Cuantificări**:  $\forall x. \alpha, \exists x. \alpha$

- + **Interpretarea** constă din:
  - Un **domeniu** nevid,  $D$ , de concepte (obiecte)
  - Pentru fiecare **constantă**  $c$ , un element  $c^I \in D$
  - Pentru fiecare simbol **funcțional**,  $n$ -ar  $f$ , o funcție  $f^I : D^n \rightarrow D$
  - Pentru fiecare **predicat**  $n$ -ar  $p$ , o funcție  $p^I : D^n \rightarrow \{false, true\}$ .

+ **Termeni** (obiecte):

- Constante;
- Variabile;
- Aplicații de funcții:  $f(t_1, \dots, t_n)$ , unde  $f$  este un simbol **funcțional**  $n$ -ar și  $t_1, \dots, t_n$  sunt termeni.

Exemple

- $succesor(4)$ : succesul lui 4, și anume 5.
- $+(2, x)$ : aplicația funcției de adunare asupra numerelor 2 și  $x$ , și, totodată, suma lor.

Exemplu

“Tatăl Ioanei are un prieten deștept”

$$\exists X. \underbrace{prien(X, \underbrace{tata}_{\text{termen}}(\underbrace{ioana}_{\text{termen}}))}_{\text{atom/propoziție}} \wedge \underbrace{deștept(X)}_{\text{atom/propoziție}}$$

Diagram illustrating the logical structure of the sentence "Tatăl Ioanei are un prieten deștept". The expression is  $\exists X. prieten(X, tata(ioana)) \wedge deștept(X)$ . Brackets identify the components:  $tata$  and  $ioana$  are terms;  $prien(X, tata(ioana))$  is an atom/proposition;  $deștept(X)$  is an atom/proposition; and the entire expression is a proposition.

- **Atom**:  $(p(t_1, \dots, t_n))^I = p^I(t_1^I, \dots, t_n^I)$
- **Negație, conectori, implicații**: v. logica propozițională
- **Cuantificare universală**:  $(\forall x. \alpha)^I = \begin{cases} false & \text{dacă } \exists d \in D. \alpha^I_{[d/x]} = false \\ true & \text{altfel} \end{cases}$
- **Cuantificare existențială**:  $(\exists x. \alpha)^I = \begin{cases} true & \text{dacă } \exists d \in D. \alpha^I_{[d/x]} = true \\ false & \text{altfel} \end{cases}$

Ex Exemple cu cuantificatori

- 1 "Vrabia mălai visează."  $\forall x.(vrabie(x) \Rightarrow viseaza(x, malai))$
- 2 "Unele vrăbii visează mălai."  $\exists x.(vrabie(x) \wedge viseaza(x, malai))$
- 3 "Nu toate vrăbiile visează mălai."  $\exists x.(vrabie(x) \wedge \neg viseaza(x, malai))$
- 4 "Nicio vrabie nu visează mălai."  $\forall x.(vrabie(x) \Rightarrow \neg viseaza(x, malai))$
- 5 "Numai vrăbiile visează mălai."  $\forall x.(viseaza(x, malai) \Rightarrow vrabie(x))$

- Satisfiabilitate.
- Validitate.
- Derivabilitate.
- Inferență.

+ **Clauză Horn** – Clauză în care cel mult un literal este în formă pozitivă:  
 $\{-A_1, \dots, -A_n, A\}$ ,  
corespunzătoare implicației  
 $A_1 \wedge \dots \wedge A_n \Rightarrow A$ .

Ex Exemplu Transformarea propoziției  
 $\forall x.vrabie(x) \vee ciocarlie(x) \Rightarrow pasare(x)$  în formă normală, utilizând clauze  
Horn:  
FNC:  $\{-vrabie(x), pasare(x)\}, \{-ciocarlie(x), pasare(x)\}$

- $\forall x.(vrabie(x) \Rightarrow viseaza(x, malai))$   
→ corect: "Toate vrăbiile visează mălai."
- $\forall x.(vrabie(x) \wedge viseaza(x, malai))$   
→ **gresit**: "Toți sunt vrăbii și toți visează mălai."
- $\exists x.(vrabie(x) \wedge viseaza(x, malai))$   
→ corect: "Unele vrăbii visează mălai."
- $\exists x.(vrabie(x) \Rightarrow viseaza(x, malai))$   
→ **gresit**: probabil nu are semnificația pe care o intenționăm. Este adevărată și dacă luăm un x care nu este vrabie (fals implică orice).

### Forme normale

- 1 Eliminarea **implicațiilor** ( $\Rightarrow$ )
- 2 Împingerea **negațiilor** până în fața atomilor ( $\neg$ )
- 3 **Redenumirea** variabilelor cuantificate pentru obținerea **unicității** de nume (R):  
 $\forall x.p(x) \wedge \forall x.q(x) \vee \exists x.r(x) \rightarrow \forall x.p(x) \wedge \forall y.q(y) \vee \exists z.r(z)$
- 4 **Deplasarea** cuantificatorilor la **începutul** expresiei, conservându-le **ordinea** (forma normală **prenex**) (P):  
 $\forall x.p(x) \wedge \forall y.q(y) \vee \exists z.r(z) \rightarrow \forall x.\forall y.\exists z.(p(x) \wedge q(y) \vee r(z))$

- **Necomutativitate**:
  - $\forall x.\exists y.viseaza(x, y) \rightarrow$  "Toți visează la câte ceva."
  - $\exists y.\forall x.viseaza(x, y) \rightarrow$  "Există ceva la care visează toată lumea."
- **Dualitate**:
  - $\neg(\forall x.\alpha) \equiv \exists x.\neg\alpha$
  - $\neg(\exists x.\alpha) \equiv \forall x.\neg\alpha$

- + **Literal** – Atom sau **negația** unui atom.
- Ex Exemplu  $prieten(x, y), \neg prieten(x, y)$ .
- + **Clauză** – Multime de literali dintr-o expresie clauzală.
- Ex Exemplu  $\{prien(x, y), \neg doctor(x)\}$ .
- + **Forma normală conjunctivă – FNC** – Reprezentare ca **mulțime de clauze**, cu semnificație conjunctivă.
- + **Forma normală implicativă – FNI** – Reprezentare ca **mulțime de clauze** cu clauzele în forma **grupată**  
 $\{\neg A_1, \dots, \neg A_m, B_1, \dots, B_n\}, \Leftrightarrow (A_1 \wedge \dots \wedge A_m) \Rightarrow (B_1 \vee \dots \vee B_n)$

- 1 Eliminarea cuantificatorilor **existențiali** (skolemizare) (S):
  - Dacă **nu** este precedat de cuantificatori universali: înlocuirea aparițiilor variabilei cuantificate printr-o **constantă** (bine aleasă):  
 $\exists x.p(x) \rightarrow p(c_x)$
  - Dacă este **precedat** de cuantificatori universali: înlocuirea aparițiilor variabilei cuantificate prin aplicația unei **funcții** unice asupra variabilelor anterior cuantificate universal:  
 $\forall x.\forall y.\exists z.((p(x) \wedge q(y)) \vee r(z))$   
 $\rightarrow \forall x.\forall y.((p(x) \wedge q(y)) \vee r(f_z(x, y)))$

## Conversia propozițiilor în FNC (3)

Cuantificatori universali, Distribuire  $\vee$ , Clauze

$P \vee \bar{P}$

- Eliminarea cuantificatorilor **universali**, considerați, acum, implicați ( $\forall$ ):

$$\forall x. \forall y. (p(x) \wedge q(y) \vee r(f_z(x, y))) \rightarrow p(x) \wedge q(y) \vee r(f_z(x, y))$$

- Distribuirea** lui  $\vee$  față de  $\wedge$  ( $\vee/\wedge$ ):

$$\alpha \vee (\beta \wedge \gamma) \rightarrow (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$$

- Transformarea expresiilor în **clauze** (C).

## Rezoluție

O metodă de inferență completă și consistentă

$P \vee \bar{P}$

- Pasul de rezoluție:** regulă de inferență foarte puternică.
- Baza unui demonstrator de teoreme **consistent și complet**.
- Spațiul de căutare mai mic decât în alte sisteme.
- Se bazează pe lucrul cu propoziții în **forma clauzală** (clauze):
  - propoziție = mulțime de **clauze** (semnificație conjunctivă)
  - clauză = mulțime de **literali** (semnificație disjunctivă)
  - literal = **atom** sau **atom negat**
  - atom = **propoziție simplă**

## Rezoluție

Demonstrare

$P \vee \bar{P}$

- Demonstrarea **nesatisfiabilității**  $\rightarrow$  derivarea clauzei **vide**.
- Demonstrarea **derivabilității** concluziei  $\phi$  din premisele  $\phi_1, \dots, \phi_n \rightarrow$  demonstrarea **nesatisfiabilității** propoziției  $\phi_1 \wedge \dots \wedge \phi_n \wedge \neg \phi$ .
- Demonstrarea **validității** propoziției  $\phi \rightarrow$  demonstrarea **nesatisfiabilității** propoziției  $\neg \phi$ .

## Conversia propozițiilor în FNC – Exemplu

$P \vee \bar{P}$

**Exemplu** “Cine rezolvă toate laboratoarele este apreciat de cineva.”

$$\forall x. (\forall y. (lab(y) \Rightarrow rezolva(x, y)) \Rightarrow \exists y. apreciaza(y, x))$$

$$\forall x. (\neg \forall y. (\neg lab(y) \vee rezolva(x, y)) \vee \exists y. apreciaza(y, x))$$

$$\forall x. (\exists y. \neg (\neg lab(y) \vee rezolva(x, y)) \vee \exists y. apreciaza(y, x))$$

$$\forall x. (\exists y. (lab(y) \wedge \neg rezolva(x, y)) \vee \exists y. apreciaza(y, x))$$

$$\forall x. (\exists y. (lab(y) \wedge \neg rezolva(x, y)) \vee \exists z. apreciaza(z, x))$$

$$\forall x. \exists y. \exists z. ((lab(y) \wedge \neg rezolva(x, y)) \vee apreciaza(z, x))$$

$$\forall x. ((lab(f_y(x)) \wedge \neg rezolva(x, f_y(x))) \vee apreciaza(f_z(x), x))$$

$$\forall x. (lab(f_y(x)) \wedge \neg rezolva(x, f_y(x))) \vee apreciaza(f_z(x), x)$$

$$\vee/\wedge (lab(f_y(x)) \vee apr(f_z(x), x)) \wedge (\neg rez(x, f_y(x)) \vee apr(f_z(x), x))$$

$$C \{lab(f_y(x)), apr(f_z(x), x)\}, \{\neg rez(x, f_y(x)), apr(f_z(x), x)\}$$

## Rezoluție

Principiu de bază  $\rightarrow$  pasul de rezoluție

$P \vee \bar{P}$



**Ideea** (în LP):

$$\frac{\{p \Rightarrow q\}}{\{p \Rightarrow r\}} \rightarrow \text{“Anularea” lui } p$$

- $p$  **falsă**  $\rightarrow \neg p$  adevărată  $\rightarrow r$  adevărată
- $p$  **adevărată**  $\rightarrow q$  adevărată
- $p \vee \neg p \Rightarrow$  **Cel puțin una** dintre  $q$  și  $r$  adevărată ( $q \vee r$ )

- Forma generală a **pasului de rezoluție**:

$$\frac{\{p_1, \dots, r, \dots, p_m\}}{\{q_1, \dots, \neg r, \dots, q_n\}} \rightarrow \{p_1, \dots, p_m, q_1, \dots, q_n\}$$

## Rezoluție

Exemplu în LP

$P \vee \bar{P}$

Demonstrăm că  $\{p \Rightarrow q, q \Rightarrow r\} \vdash p \Rightarrow r$ ,  
i.e. mulțimea  $\{p \Rightarrow q, q \Rightarrow r, \neg(p \Rightarrow r)\}$  conține o **contradicție**.

Exemplu

- $\{ \neg p, q \}$  Premisă
- $\{ \neg q, r \}$  Premisă
- $\{ p \}$  Concluzie negată
- $\{ \neg r \}$  Concluzie negată
- $\{ q \}$  Rezoluție 1, 3
- $\{ r \}$  Rezoluție 2, 5
- $\{ \}$  Rezoluție 4, 6  $\rightarrow$  clauza vidă

## Unificare și rezoluție

## Rezoluție

Cazuri speciale

$P \vee \bar{P}$

- Clauza **vidă**  $\rightarrow$  indicator de **contradicție** între premise

$$\frac{\{\neg p\}}{\{p\}} \rightarrow \{\} = \emptyset$$

- Mai mult de 2** rezolvenți posibili  $\rightarrow$  se alege doar unul:

$$\frac{\{p, q\}}{\{\neg p, \neg q\}} \rightarrow \{p, \neg p\} \text{ sau } \{q, \neg q\}$$

## Rezoluție

Consistentă și completitudine

$P \vee \bar{P}$

**T Teorema Rezoluției:** Rezoluția propozițională este **consistentă și completă**, i.e.  $\Delta \models \phi \Leftrightarrow \Delta \vdash_{rez} \phi$ .

- Terminare garantată** a procedurii de aplicare a rezoluției: număr **finit** de clauze  $\rightarrow$  număr **finit** de concluzii.

## Unificare

P ∨ P

- Utilizată pentru rezoluția în LPOI
- vezi și sinteza de tip în Haskell

cum știm dacă folosind ipoteza  $om(Marcel)$  și propoziția  $\forall x.om(x) \Rightarrow are\_inima(x)$  putem demonstra că  $are\_inima(Marcel) \rightarrow$  unificând  $om(Marcel)$  și  $\forall om(x)$ .

- reguli:
  - o propoziție unifică cu o propoziție de aceeași formă
  - două predicate unifică dacă au același nume și parametri care unifică ( $om$  cu  $om$ ,  $x$  cu  $Marcel$ )
  - o constantă unifică cu o constantă cu același nume
  - o variabilă unifică cu un termen ce nu conține variabila ( $x$  cu  $Marcel$ )

Logica propozițională Evaluare LPOI LPOI – Semantică Logica cu predicate de ordinul I Forme normale Unificare și rezoluție 10 : 48

## Rezoluție

Exemplu

P ∨ P

Horses and hounds

- 1 Horses are faster than dogs.
- 2 There is a greyhound that is faster than any rabbit.
- 3 Harry is a horse and Ralph is a rabbit.
- 4 Is Harry faster than Ralph?

Logica propozițională Evaluare LPOI LPOI – Semantică Logica cu predicate de ordinul I Forme normale Unificare și rezoluție 10 : 51

## Introducere în Prolog

Introducere în Prolog Demonstrare Introducere în Prolog Controlul execuției 11 : 2

## Unificare

Observații

P ∨ P

- Problemă NP-completă;
- Posibile legări ciclice;
- Exemplu:  
 $prieten(x, coleg\_banca(x))$  și  
 $prieten(coleg\_banca(y), y)$   
MGU:  $S = \{x \leftarrow coleg\_banca(y), y \leftarrow coleg\_banca(x)\}$   
 $\Rightarrow x \leftarrow coleg\_banca(coleg\_banca(x)) \rightarrow$  imposibil!
- Soluție: verificarea apariției unei variabile în valoarea la care a fost legată (occurrence check);

Logica propozițională Evaluare LPOI LPOI – Semantică Logica cu predicate de ordinul I Forme normale Unificare și rezoluție 10 : 49

## Rezoluție

Exemplu Horses and Hounds

P ∨ P

- 1  $\forall x.\forall y.horse(x) \wedge dog(y) \Rightarrow faster(x,y) \rightarrow \neg horse(x) \vee \neg dog(y) \vee faster(x,y)$
- 2  $\exists x.greyhound(x) \wedge (\forall y.rabbit(y) \Rightarrow faster(x,y))$   
 $\rightarrow greyhound(Greg) ; \neg rabbit(y) \vee faster(Greg,y)$
- 3  $horse(Harry) ; rabbit(Ralph)$
- 4  $\neg faster(Harry, Ralph)$  (concluzia negată)
- 5  $\neg greyhound(x) \vee dog(x)$  (common knowledge)
- 6  $\neg faster(x,y) \vee \neg faster(y,z) \vee faster(x,z)$  (tranzitivitate)
- 7  $1 + 3a \rightarrow \neg dog(y) \vee faster(Harry,y)$  (cu  $\{Harry/x\}$ )
- 8  $2a + 5 \rightarrow dog(Greg)$  (cu  $\{Greg/x\}$ )
- 9  $7 + 8 \rightarrow faster(Harry,Greg)$  (cu  $\{Greg/y\}$ )
- 10  $2b + 3b \rightarrow faster(Greg,Ralph)$  (cu  $\{Ralph/y\}$ )
- 11  $6 + 9 + 10 \rightarrow faster(Harry,Ralph)$  ( $\{Harry/x, Greg/y, Ralph/z\}$ )
- 12  $11 + 4 \rightarrow \square$  q.e.d.

Logica propozițională Evaluare LPOI LPOI – Semantică Logica cu predicate de ordinul I Forme normale Unificare și rezoluție 10 : 52

## Prolog

Limbaj de programare logică



- introdus în anii 1970 ;
- programul  $\rightarrow$  mulțime de propoziții logice în LPOI;
- mediul de execuție = demonstrator de teoreme care spune:
  - dacă un fapt este adevărat sau fals;
  - în ce condiții este un fapt adevărat.

- Resursă Prolog pe Wikibooks:  
[<https://en.wikibooks.org/wiki/Prolog>]

Introducere în Prolog Demonstrare Introducere în Prolog Controlul execuției 11 : 3

## Unificare

Rolul în rezoluție

P ∨ P

- Rezoluția pentru clauze Horn:  
 $A_1 \wedge \dots \wedge A_m \Rightarrow A$   
 $B_1 \wedge \dots \wedge A' \wedge \dots \wedge B_n \Rightarrow B$   
 $\frac{unificare(A, A')}{subst(S, A_1 \wedge \dots \wedge A_m \wedge B_1 \wedge \dots \wedge B_n \Rightarrow B)}$
- $unificare(\alpha, \beta) \rightarrow$  substituția sub care unifică propozițiile  $\alpha$  și  $\beta$ ;
- $subst(S, \alpha) \rightarrow$  propoziția rezultată în urma aplicării substituției  $S$  asupra propoziției  $\alpha$ .

Logica propozițională Evaluare LPOI LPOI – Semantică Logica cu predicate de ordinul I Forme normale Unificare și rezoluție 10 : 50

## Cursul 11: Introducere în Prolog



- 45 Introducere în Prolog
- 46 Procesul de demonstrare
- 47 Controlul execuției

Introducere în Prolog Demonstrare Introducere în Prolog Controlul execuției 11 : 1

## Prolog

Caracteristici



- fundamentare teoretică a procesului de raționament;
- motor de raționament ca unic mod de execuție;  
 $\rightarrow$  modalități limitate de control al execuției.
- căutare automată a valorilor pentru variabilele nelegate (dacă este necesar);
- posibilitatea demonstrațiilor și deducțiilor simbolice.

Introducere în Prolog Demonstrare Introducere în Prolog Controlul execuției 11 : 4

## Procesul de demonstrare

### Pași în demonstrare (1)

- 1 Inițializarea **stivei de scopuri** cu scopul solicitat;
- 2 Inițializarea **substituției** (utilizate pe parcursul unificării) cu mulțimea vidă;
- 3 Extragerea scopului din **vârful** stivei și determinarea **primei** clauze din program cu a cărei concluzie **unifică**;
- 4 Îmbogățirea corespunzătoare a **substituției** și adăugarea **premiselor** clauzei în stivă, în ordinea din program;
- 5 Salt la pasul 3.

### Pași în demonstrare (2)

- 6 În cazul **imposibilității** satisfacerii scopului din vârful stivei, **revenirea** la scopul anterior (*backtracking*), și încercarea altei modalități de satisfacere;
- 7 **Succes** la **golirea** stivei de scopuri;
- 8 **Eșec** la imposibilitatea satisfacerii **ultimului** scop din stivă.

## Un exemplu de program Prolog

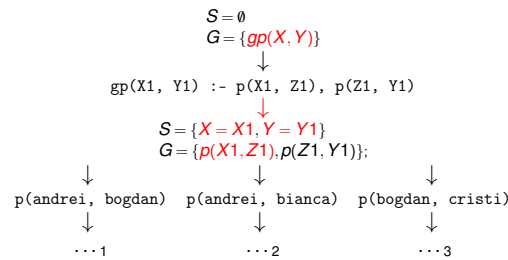
### Exemplu

```

1 parent(andrei, bogdan).
2 parent(andrei, bianca).
3 parent(bogdan, cristi).
4
5 grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
    
```

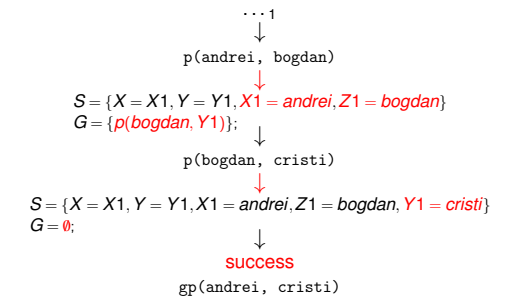
- $true \Rightarrow parent(andrei, bogdan)$
- $true \Rightarrow parent(andrei, bianca)$
- $true \Rightarrow parent(bogdan, cristi)$
- $\forall x, \forall y, \forall z. (parent(x, z) \wedge parent(z, y) \Rightarrow grandparent(x, y))$

## Exemplul genealogic (1)



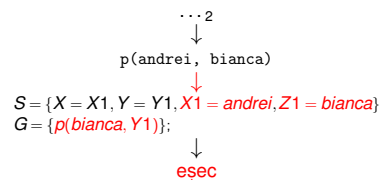
## Exemplul genealogic (2)

### Ramura 1



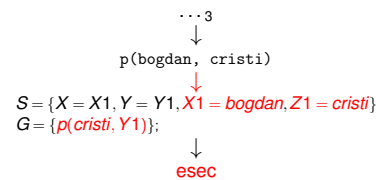
## Exemplul genealogic (3)

### Ramura 2



## Exemplul genealogic (4)

### Ramura 3



## Observații

- Ordinea evaluării / încercării demonstrării scopurilor
  - Ordinea **clauzelor** în program;
  - Ordinea **premiselor** în cadrul regulilor.
- Recomandare: premisele **mai ușor** de satisfăcut și **mai specifice** primele – exemplu: axiome.

## Strategii de control

Ale demonstrațiilor



### Forward chaining (data-driven)

- Derivarea **tuturor** concluziilor, pornind de la datele initiale;
- **Oprire** la obținerea scopului (scopurilor);

### Backward chaining (goal-driven)

- Utilizarea **exclusivă** a regulilor care pot contribui efectiv la satisfacerea scopului;
- Determinarea regulilor a căror concluzie **unifică** cu scopul;
- Încercarea de satisfacere a **premiselor** acestor reguli ș.a.m.d.

Introducere în Prolog

Demonstrare  
Introducere în Prolog

Controlul execuției 11 : 14

## Exemplu – Minimul a două numere

Cod Prolog



### Exemplu | Minimul a două numere

```
1 min(X, Y, M) :- X =< Y, M is X.
2 min(X, Y, M) :- X > Y, M is Y.
3
4 min2(X, Y, M) :- X =< Y, M = X.
5 min2(X, Y, M) :- X > Y, M = Y.
6
7 % Echivalent cu min2.
8 min3(X, Y, X) :- X =< Y.
9 min3(X, Y, Y) :- X > Y.
```

Introducere în Prolog

Demonstrare  
Introducere în Prolog

Controlul execuției 11 : 17

## Exemplu – Minimul a două numere

Îmbunătățire



- Soluție: **oprirea** recursivității după prima satisfacere a scopului.

### Exemplu

```
1 min5(X, Y, X) :- X =< Y, !.
2 min5(X, Y, Y).
3
4 ?- min5(1+2, 3+4, M).
5 M = 1+2.
```

Introducere în Prolog

Demonstrare  
Introducere în Prolog

Controlul execuției 11 : 20

## Strategii de control

Algoritm Backward chaining



1. **BackwardChaining**(rules, goals, subst)  
lista **regulilor** din program, stiva de **scopuri**, substituția curentă, inițial vidă.  
**returns** satisfiabilitatea scopurilor
2. **if** goals =  $\emptyset$  **then**
3.     **return** SUCCESS
4.     goal  $\leftarrow$  head(goals)
5.     goals  $\leftarrow$  tail(goals)
6.     **for-each** rule  $\in$  rules **do** // în ordinea din program
7.         **if** unify(goal, conclusion(rule), subst)  $\rightarrow$  bindings
8.         newGoals  $\leftarrow$  premises(rule)  $\cup$  goals // **adâncime**
9.         newSubst  $\leftarrow$  subst  $\cup$  bindings
10.        **if** BackwardChaining(rules, newGoals, newSubst)
11.        **then return** SUCCESS
12. **return** FAILURE

Introducere în Prolog

Demonstrare  
Introducere în Prolog

Controlul execuției 11 : 15

## Exemplu – Minimul a două numere

Utilizare



```
1 ?- min(1+2, 3+4, M).
2 M = 3 ;
3 false.
4
5 ?- min(3+4, 1+2, M).
6 M = 3.
7
8 ?- min2(1+2, 3+4, M).
9 M = 1+2 ;
10 false.
11
12 ?- min2(3+4, 1+2, M).
13 M = 1+2.
```

Introducere în Prolog

Demonstrare  
Introducere în Prolog

Controlul execuției 11 : 18

## Operatorul cut

Definiție



- La **prima** întâlnire  $\rightarrow$  **satisfacere**;
- La **a doua** întâlnire în momentul revenirii (**backtracking**)  $\rightarrow$  **eșec**, cu inhibarea **tuturor** căilor ulterioare de satisfacere a scopului care a unificat cu concluzia regulii curente;
- Utilitate în **eficientizarea** programelor.

Introducere în Prolog

Demonstrare  
Introducere în Prolog

Controlul execuției 11 : 21

## Controlul execuției

## Exemplu – Minimul a două numere

Observații



- Condiții mutual exclusive:  $X =< Y$  și  $X > Y \rightarrow$  cum putem **elimina** redundanța?

### Exemplu

```
1 min4(X, Y, X) :- X =< Y.
2 min4(X, Y, Y).
3
4 ?- min4(1+2, 3+4, M).
5 M = 1+2 ;
6 M = 3+4.
```

- **Greșit!**

Introducere în Prolog

Demonstrare  
Introducere în Prolog

Controlul execuției 11 : 19

## Operatorul cut

Exemplu



### Exemplu

```
1 girl(mary).
2 girl(ann).
3
4 boy(john).
5 boy(bill).
6
7 pair(X, Y) :- girl(X), boy(Y).
8 pair(bella, harry).
9
10 pair2(X, Y) :- girl(X), !, boy(Y).
11 pair2(bella, harry).
```

Introducere în Prolog

Demonstrare  
Introducere în Prolog

Controlul execuției 11 : 22



```
1 ?- pair(X, Y).
2 X = mary,
3 Y = john ;
4 X = mary,
5 Y = bill ;
6 X = ann,
7 Y = john ;
8 X = ann,
9 Y = bill ;
10 X = bella,
11 Y = harry.
```

```
1 ?- pair2(X, Y).
2 X = mary,
3 Y = john ;
4 X = mary,
5 Y = bill.
```



### Exemplu

```
1 nott(P) :- P, !, fail.
2 nott(P).
```

- P: atom – exemplu: boy(john)
- dacă P este **satisfiabil**:
  - eșecul **primei** reguli, din cauza lui `fail`;
  - abandonarea celei **de-a doua** reguli, din cauza lui `!`;
  - rezultat: `nott(P)` **nesatisfiabil**.
- dacă P este **nesatisfiabil**:
  - eșecul **primei** reguli;
  - succesul celei **de-a doua** reguli;
  - rezultat: `nott(P)` **satisfiabil**.